# The Variadic Macro Data Library 1.8

Edward Diener



## Table of Contents

# Introduction

Welcome to version 1.8 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, types, tuples, arrays, lists, and seqs.

3. Provide ways for testing/parsing sequences of identifiers, numbers, types, 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.

# 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
7. TYPE = A VMD type

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 'array', 'list', 'seq', and 'tuple' refer to these Boost PP data types unless otherwise noted. See the help for Boost PP for any explanation of these data types.

The term 'sequence' refers to a sequence of VMD data types and is not the same as a Boost PP sequence which is always referred to in this documentation as a 'seq'.

The term 'return' refers to the expansion of a macro. I use the terminology of a macro "returning some data" rather than the terminology of a macro "expanding to some data", even if the latter is more accurate, because it more closely corresponds to the way I believe C++ programmers think about macro programming.

The term 'emptiness' refers to no preprocessor data being passed to or returned from a macro. I have avoided the word 'nothing' because it has too vague a meaning.

The term 'data type' refers to the various preprocessor input types which VMD can parse and which are listed above, also including emptiness.

The term 'v-type' refers to a VMD type, the term 'number' returns to a VMD number and the term 'identifier' refers to a VMD identifier. All these will be explained in their proper place.

The term "UB" stands for "undefined behavior" as it is specified in the C++ standard.

# Why and how to use

The VMD library provides the ability to create a macro which takes different types of parameters, and generates different output depending on the parameter types as well as their values.

This is equivalent to the way that overloaded functions provide the ability for a singularly named function to provide different functionality depending on the parameter types.

In the case of macros, where more than one macro of the same name but different macro expansion is not allowed, a single macro name can create different expansions.

As a simple example:

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

#define AMACRO(param)                 \
  BOOST_PP_IIF                        \
    (                                 \
    BOOST_VMD_IS_SEQ(param),          \
    Seq,                              \
    BOOST_PP_IIF                      \
      (                               \
      BOOST_VMD_IS_TUPLE(param),      \
      Tuple,                          \
      Unknown                         \
      )                               \
    )
```

If the param passed is a seq the output of the macro is 'Seq'. If the param passed is a tuple the output of the macro is 'Tuple'. Otherwise the output of the macro is 'Unknown'.

Obviously much more complicated cases can be created in which the types and values of various parameters are parsed, in order to produce variable macro output depending on the input. Using variadic macros, macros with variable numbers and types of arguments give the macro programmer even greater freedom to design macros with flexibility.

Another feature of the VMD library is the ability to parse identifiers. A system of registering identifiers which VMD can recognize has been created. Once an identifier is registered VMD can recognize it as part of macro input as an identifier and return the identifier. Furthermore VMD can compare identifiers for equality or inequality once an identifier has been pre-detected using VMD's system for pre-detecting identifiers.

As another simple example:

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

#define BOOST_VMD_REGISTER_NAME (NAME)
#define BOOST_VMD_REGISTER_ADDRESS (ADDRESS)

#define AMACRO1(param)              \
  BOOST_PP_IIF                      \
    (                               \
    BOOST_VMD_IS_IDENTIFIER(param), \
    AMACRO1_IDENTIFIER,             \
    AMACRO1_NO_IDENTIFIER           \
    )                               \
  (param)

#define AMACRO1_IDENTIFIER(param) AMACRO1_ ## param
#define AMACRO1_NO_IDENTIFIER(param) Parameter is not an identifier
#define AMACRO1_NAME Identifier is a NAME
#define AMACRO1_ADDRESS Identifier is an ADDRESS
```

Here we use VMD's identifier registration system to determine and handle a particular identifier we may be expecting as a macro parameter.

Identifier pre-detection makes things clearer, allowing us to detect within VMD whether macro input matches a particular identifier. Using the same setup as our previous example, but with identifier pre-detection:

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

#define BOOST_VMD_REGISTER_NAME (NAME)
#define BOOST_VMD_DETECT_NAME_NAME

#define BOOST_VMD_REGISTER_ADDRESS (ADDRESS)
#define BOOST_VMD_DETECT_ADDRESS_ADDRESS

#define AMACRO2(param)                          \
  BOOST_PP_IIF                                  \
    (                                           \
    BOOST_VMD_IS_IDENTIFIER(param,NAME),        \
    AMACRO2_NAME,                               \
    BOOST_PP_IIF                                \
      (                                         \
      BOOST_VMD_IS_IDENTIFIER(param,ADDRESS),   \
      AMACRO2_ADDRESS,                          \
      AMACRO2_NO_IDENTIFIER                     \
      )                                         \
    )                                           \
  (param)

#define AMACRO2_NO_IDENTIFIER(param) Parameter is not a NAME or ADDRESS identifier
#define AMACRO2_NAME(param) Identifier is a NAME
#define AMACRO2_ADDRESS(param) Identifier is an ADDRESS
```

The VMD library also has 2 different subtypes of identifiers which can always be recognized. The first are numbers, equivalent to the number in Boost PP, numeric values with a range of 0-256. The second are v-types, which are identifiers starting with BOOST_VMD_TYPE_ followed by a name for the type of data. As an example, the v-type of a Boost PP tuple is BOOST_VMD_TYPE_TUPLE and the v-type of a v-type itself is BOOST_VMD_TYPE_TYPE. All data types have their own v-type identifier; types are recognized by the VMD macros and may be passed as input data just like any other of the types of data VMD recognizes.

The VMD identifier system even has a way, to be explained later, for the end-user to create his own subtype identifiers.

Another reason to use VMD is that VMD understands 'sequences' of the VMD data types. You can have a sequence of data types and VMD can convert the sequence to any of the Boost PP data types, or access any individual data type in a sequence.

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

#define BOOST_VMD_REGISTER_NAME (NAME)
#define ASEQUENCE (1,2) NAME 147 BOOST_VMD_TYPE_NUMBER (a)(b)

BOOST_VMD_TO_TUPLE(ASEQUENCE)
BOOST_VMD_ELEM(2,ASEQUENCE)
```

Our first expansion returns the tuple:

```
((1,2),NAME,147,BOOST_VMD_TYPE_NUMBER,(a)(b))
```

Our second expansion returns the sequence element:

```
147
```

Sequences give the macro programmer the ability to accept input data from the user which may more closely mimic C++ constructs.

Another reason to use VMD is that VMD understands data types. Besides specifically asking if a particular input is a particular data type, you can use the macro BOOST_VMD_GET_TYPE to retrieve the type of any VMD data.

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

BOOST_VMD_GET_TYPE((1,2)) // expands to BOOST_VMD_TYPE_TUPLE
BOOST_VMD_GET_TYPE(235)   // expands to BOOST_VMD_TYPE_NUMBER
```

etc.

There is still much more of VMD functionality but hopefully this brief introduction of what VMD can do will interest you so that you will read on to understand VMD's functionality for the macro programmer.

# Using 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.

# Visual C++ define

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. You can use this macro in your own macro code whenever you include a VMD header file to write code which may need special processing for VC++ as outlined in this documentation. Your macro processing may therefore occasional take the form of:

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

#if BOOST_VMD_MSVC

#define SOME_MACRO ... code for VC++

#else

#define SOME_MACRO ... code for all other compilers

#endif
```

# Functional groups

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

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

The first six categories delineate the data types which VMD can parse. The last category presents additional macros which will prove helpful for a macro programmer using variadic macros with VMD and Boost PP.

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

Furthermore VMD macros for working with the above data types which VMD understands can be divided into specific and generic macros.

The specific macros ask whether some input data is a particular data type. The generic macros work with input data as any data type while allowing the programmer to separately query the type of data.

Both specific and generic macros have their place and the macro programmer can decide which to use for any given situation.

# Data types

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

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. The preprocessor token types which VMD cannot parse are:

- header-name
- character-literal
- user-defined-characteral-literal
- string-literal
- user-defined-string-literal
- preprocessing-op-or-punc

Even though VMD cannot parse these preprocessor token types, it is still a very useful library since a large part of macro programming works with 'identifier' and 'pp-number' tokens.

VMD identifiers are 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 VMD identifier can start with a numeric character, allowing VMD identifiers to also be positive integral literals. VMD offers functionality for parsing VMD identifiers both as a separate element or in a sequence of preprocessing tokens.

VMD numbers are Boost PP numbers, ie. preprocessing tokens of whole numbers between 0 and 256 inclusive. These are a small subset of preprocessor token "pp-number". VMD offers functionality for parsing numbers both as a separate element or in a sequence of preprocessing tokens. A VMD number is really a subset of VMD identifiers for which VMD offers specific functionality. The Boost PP library has it own extensive support for numbers, which VMD does not duplicate.

VMD v-types are, like numbers, a subset of VMD identifiers consisting of identifiers beginning with BOOST_VMD_TYPE_ followed by a data type mnemonic. Each type can be recognized by VMD functionality and therefore passed or returned by macros. Like any identifier a v-type can be parsed both as a separate element or in a sequence of preprocessing tokens.

VMD can also test for emptiness, or the absence of any preprocessing tokens passed as macro input.

The Boost PP library supports four individual high-level data types. These are arrays, lists, seqs, and tuples. When using variadic macros arrays are really obsolete since tuples have all the functionality of arrays with a simpler syntax. Nonetheless arrays are fully supported by VMD. A further data type supported by Boost PP is variadic data, which is a comma separated grouping of preprocessor elements. VMD has no special support for variadic data outside of what is already in Boost PP.

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.

VMD can also parse sequences. A sequence consists of zero or more other top-level data types already mentioned represented consecutively. As such a sequence represents any data type which VMD can parse since it can consist of emptiness, a single data type, or multiple data types represented consecutively.

Emptiness, the three identifier types, the four Boost PP composite data types, and VMD sequences are the data types which VMD understands. Other low-level preprocessor data types can of course be used in macro programming but VMD cannot parse such preprocessor data.

# Specific macros for working with data types

VMD has a number of specific macros for parsing data types. Each of these macros asks if some input is a particular VMD data type.

# 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.

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 has normally been 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 not necessarily 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 as 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 if the input is actually empty.

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 always 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:

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

#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

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

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.

Another situation where the macro may not work properly is if the previously mentioned function-like macro takes a single parameter but creates an error when the argument passed is empty. An example of this would be:

```
#define FMACRO(x) BOOST_PP_CAT(+,x C);
```

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

When nothing is passed to FMACRO undefined behavior will occur.

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. Essentially what is desired for maximum safety is that we never pass input ending with the name of a function-like macro name when testing for emptiness.

## 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.

```
#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, and the function-like macro, when passed a single empty argument, expands to a tuple, BOOST_VMD_IS_EMPTY will erroneously return 1 when using the Visual C++ compiler rather than either give a preprocessing error or return 0.

Here is an example of the failure:

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

#define FMACRO4() ( any_number_of_tuple_elements )
#define FMACRO5(param) ( any_number_of_tuple_elements )
#define FMACRO6(param1,param2) ( any_number_of_tuple_elements )

BOOST_VMD_IS_EMPTY(FMACRO4) // erroneously returns 1, instead of 0
BOOST_VMD_IS_EMPTY(FMACRO5) // erroneously returns 1, instead of 0
BOOST_VMD_IS_EMPTY(FMACRO6) // erroneously returns 1, instead of generating a preprocessing error
```

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 BOOST_VMD_IS_EMPTY 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 BOOST_VMD_IS_EMPTY 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 internally 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.

You can use the general header file:

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

or you can use the individual header file:

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

for the BOOST_VMD_IS_EMPTY 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 preprocessing 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 could only occur with a preprocessor ( Visual C++ ) which is not C++ standard conformant.

The Boost PP library does have a way of generating a preprocessing 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 preprocessing 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 creating a preprocessing error or undefined behavior.

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.

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 preprocessing 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.

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.

# Identifiers

An identifier in VMD is either of two lower-level preprocessor 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
```

## Problem testing any identifier

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

and if the first character is not the left parenthesis of a tuple the remaining characters must be alphanumeric until a space character or end of input occurs.

If this is not the case the behavior is undefined, and most likely 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 anything : can be tested, identifier followed by space character

%_anything : undefined behavior and most likely a preprocessing error due to the constraint
(_anything : undefined behavior and most likely a preprocessing error due to the con↵
straint, since a single '(' does not form a tuple
44.3 : undefined behavior and most likely a preprocessing error due to the con↵
straint since '.' is not alphanumeric
```

## Identifying an identifier

In VMD the only way an identifier can be identified in preprocessor input is by a process called registration. In order to 'register' an identifier to be recognized by VMD the end-user must create, for every identifier to be recognized, an object-like macro whose form is:

```
#define BOOST_VMD_REGISTER_identifier (identifier)
```

where 'identifier' is a particular identifier we wish to identify. This is called in VMD a registration macro.

It is recommended that such registration macros be created in a header file which can be included before the end-user uses the identifier macros of VMD.

If a particular registration macro occurs more than once it is not a preprocessing error, so duplicating a registration macro will not lead to any problems since each registration macro of the same name will have the exact same object-like macro expansion.

Within a given translation unit it could potentially happen that registration macros have been included by header files which a particular end-user of VMD has not created. This should also not lead to particular problems since registration is a process for adding identifiers for any particular transalation unit. As we shall see VMD has macros for not only finding any identifier in preprocessor input but for also finding particular identifiers in preprocessor input.

## Testing for an identifier macro

The macro used to test for an identifier in VMD is called BOOST_VMD_IS_IDENTIFIER. The macro takes one required parameter which is the input against which to test.

When we invoke BOOST_VMD_IS_IDENTIFIER it returns 1 if the input represents any registered identifier, otherwise it returns 0.

As an example:

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

#define BOOST_VMD_REGISTER_yellow (yellow)
#define BOOST_VMD_REGISTER_green  (green)
#define BOOST_VMD_REGISTER_blue   (blue)

BOOST_VMD_IS_IDENTIFIER(some_input) // returns 1 if 'some_input is 'yellow','green', or 'blue'
BOOST_VMD_IS_IDENTIFIER(some_input) // returns 0 if 'some_input' is 'purple'
```

Essentially only registered identifiers can be found in VMD as identifiers.

## Detecting a particular identifier

Although registering an identifier allows VMD to recognize the string of characters as a VMD identifier, the ability to detect a particular identifier needs the end-user to specify another macro:

```
BOOST_VMD_DETECT_identifier_identifier
```

where 'identifier' is a particular identifier we wish to detect. This object-like macro expands to no output.

Like the registration macro multiple detection macros of the same identifier in a translation unit does not cause a compiler problem since the exact same object-like macro occurs.

The term for creating this macro is that we have potentially 'pre-detected' the identifier and I will use the term pre-detected as the process of creating the BOOST_VMD_DETECT macro.

The ability to detect that a VMD identifier is a particular identifier is used in VMD macros when data is compared for equality/inequality as well as when we want to match an identifier against a set of other identifiers. These situations will be explained later in the documentation when the particular macro functionality is discussed. If the programmer never uses the functionality which these situations encompass there is no need to use pre-detection for a registered identifier.

## Parsing identifiers and undefined behavior

The technique for parsing identifiers, once it is determined that the input being parsed does not begin with a set of parentheses, uses preprocessor concatenation in its parsing. This technique involves the preprocessor '##' operator to concatenate input, and examine the results of that concatenation.

When preprocessor concatenation is used the result of the concatenation must be a valid preprocessing token, else the behavior of the preprocessor is undefined. In C++ 'undefined behavior' in general means that anything can happen. In practical use when preprocessor concatenation does not produce a valid preprocessing token, a compiler is most likely to generate a preprocessing error. If the compiler chooses not to issue a preprocessing error the outcome will always mean that parsing an identifier will fail. But because the outcome is undefined behavior there is no absolute way that the programmer can determine what the outcome will be when preprocessor concatemation is used and the input being parsed contains preprocessor input which does not meet the constraints for parsing an identifier mentioned at the beginning of this topic.

In this documentation I will be using the abbreviation 'UB' as the shortened form of 'undefined behavior' to denote the particular occurence where VMD attempts to parse preprocessor input using preprocessor concatenation and undefined behavior will occur.

### 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 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 numbers are:

```
0
127
33
254
18
```

but not:

```
033
06
009
00
```

### Problem testing any number

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

and if the first character is not the left parenthesis of a tuple the remaining characters must be alphanumeric until a space character or end of input occurs.

If this is not the case the behavior is undefined, and most likely 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 anything : can be tested, identifier followed by space character

%_anything : undefined behavior and most likely a preprocessing error due to the constraint
(_anything : undefined behavior and most likely a preprocessing error due to the con↵
straint, since a single '(' does not form a tuple
44.3 : undefined behavior and most likely a preprocessing error due to the con↵
straint since '.' is not alphanumeric
```

## Testing for a number macro

The macro used to test for any 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 macro returns 0.

The Boost PP library has a great amount of functionality for working with numbers, so once you use VMD to parse/test for a number you can use Boost PP to work with that number in various ways. The VMD makes no attempt to duplicate the functionality of numbers that in the Boost PP library.

Any number is also an identifier, which has been registered and pre-detected, so you can also use the VMD functionality which works with identifiers to work with a number as an identifier if you like.

## Example

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

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

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 of 0-256
if input = %44, does not meet the constraint therefore undefined behavior
if input = 44.0, does not meet the constraint therefore undefined behavior
if input = ( 44 ), 0 since the macro begins with a tuple and this can be tested for
```

## Usage

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>
```

# Types

A subset of identifiers is VMD types, called a 'v-type'. These are identifiers which represent all of the preprocessor data types which VMD can parse. This subset of identifiers is automatically registered and pre-detected by VMD. Each identifier type begins with the unique prefix 'BOOST_VMD_TYPE_'.

The actual types are:

- BOOST_VMD_TYPE_EMPTY, represents emptiness, ie. "empty data"
- BOOST_VMD_TYPE_ARRAY, a Boost PP array
- BOOST_VMD_TYPE_LIST, a Boost PP list
- BOOST_VMD_TYPE_SEQ, a Boost PP seq
- BOOST_VMD_TYPE_TUPLE, a Boost PP tuple
- BOOST_VMD_TYPE_IDENTIFIER, identifier
- BOOST_BMD_TYPE_NUMBER, a number
- BOOST_VMD_TYPE_TYPE, a type itself
- BOOST_VMD_TYPE_SEQUENCE, a sequence
- BOOST_VMD_TYPE_UNKNOWN, an unknown type

Since a v-type is itself an identifier the particular constraint on the input to test is exactly the same as for 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

and if the first character is not the left parenthesis of a tuple the remaining characters must be alphanumeric until a space character or end of input occurs.

If this is not the case the behavior is undefined, and most likely 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
BOOST_VMD_TYPE_NUMBER : can be tested
BOOST_VMD_TYPE_TUPLE245e2: can be tested
(anything) : can be tested, tuple
(anything) anything : can be tested, tuple + identifier
anything anything : can be tested, identifier followed by space character

%_anything : undefined behavior and most likely a preprocessing error due to the constraint
(_anything : undefined behavior and most likely a preprocessing error due to the con↵
straint, since a single '(' does not form a tuple
44.3 : undefined behavior and most likely a preprocessing error due to the con↵
straint since '.' is not alphanumeric
```

The macro used to test for a particular type in VMD is called BOOST_VMD_IS_TYPE. The macro takes a single parameter, the input to test against.

The macro returns 1 if the parameter is a v-type, otherwise the macro returns 0.

A v-type is also an identifier, which has been registered and pre-detected, so you can also use the VMD functionality which works with identifiers to work with a v-type as an identifier if you like.

### Example

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

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

BOOST_VMD_IS_TYPE(input)

returns:

if input = BOOST_VMD_TYPE_SEQ, 1
if input = BOOST_VMD_TYPE_NUMBER, 1
if input = SQUARE, 0
if input = BOOST_VMD_TYPE_IDENTIFIER DATA, 0 since there are tokens after the type
if input = %44, does not meet the constraint therefore undefined behavior
if input = ( BOOST_VMD_TYPE_EMPTY ), 0 since the macro be↵
gins with a tuple and this can be tested for
```

### Usage

To use the BOOST_VMD_IS_TYPE macro either include the general header:

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

or include the specific header:

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

# VMD and 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.

## Syntax anomalies

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

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

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
```

A list whose first element is the number 2 and whose second element is not the end-of-list marker BOOST_PP_NIL is also an array. So if one has:

```
#define ALIST (2,(3,BOOST_PP_NIL))
#define ALIST2 (2,(3,(4,BOOST_PP_NIL)))
#define ALIST3 (2,BOOST_PP_NIL)

#include <boost/vmd/is_array.hpp>
#include <boost/vmd/is_list.hpp>

BOOST_VMD_IS_LIST(ALIST) returns 1
BOOST_VMD_IS_LIST(ALIST2) returns 1
BOOST_VMD_IS_LIST(ALIST3) returns 1
BOOST_VMD_IS_ARRAY(ALIST) returns 1
BOOST_VMD_IS_ARRAY(ALIST2) returns 1
BOOST_VMD_IS_ARRAY(ALIST3) returns 0
```

A single element tuple is also a one element seq. So if one has:

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

then

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

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

## Problem when testing an array

The form of an array is a two element tuple, where the first element is a number and the second element is a tuple. The number specifies the size of the tuple. Since when using variadic macros it is never necessary to specify the size of a tuple an array is largely obsolete. However VMD still supports it.

The problem when testing for an array is that if the first element does not obey the constraint on testing for a number, you will get UB.

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

#define A_TUPLE (&anything,(1,2))

BOOST_VMD_IS_ARRAY(A_TUPLE) will give UB due to the constraint
BOOST_VMD_IS_TUPLE(A_TUPLE) will return 1
```

When VMD attempts to parse for an array, as it does when the BOOST_VMD_IS_ARRAY is used, if first looks to see if the syntax represents a tuple with two elements. Next it looks to see if the second element itself is a tuple. Finally if it is satisfied that the previous checks are valid it tests whether the first element is a number or not. It is in this final test, that the first element is a valid number, where the UB could occur as explained in the topic 'Numbers'.

## Problem when testing a list

The form of a non-empty list is a two element tuple, where the first element is the head of the list and can be anything and the second element is itself a list or the end-of-list identifier BOOST_PP_NIL.

The problem when testing for a list is that if the second element does not obey the constraint on testing for an identifier, you will get UB.

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

#define A_TUPLE (element,&anything)

BOOST_VMD_IS_LIST(A_TUPLE) will give UB due to the constraint
BOOST_VMD_IS_TUPLE(A_TUPLE) will return 1
```

The form of an empty list is the identifier BOOST_PP_NIL. Therefore:

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

#define A_BAD_EMPTY_LIST &BOOST_PP_NIL

BOOST_VMD_IS_LIST(A_BAD_EMPTY_LIST) will give UB due to the constraint
BOOST_VMD_IS_IDENTIFIER(A_BAD_EMPTY_LIST) will give UB due to the constraint
```

When VMD attempts to parse for a list, as it does when the BOOST_VMD_IS_LIST is used, if first looks to see if the syntax represents a tuple with two elements. If it is not a tuple with two elements it will check for the end-of-list. If it is a tuple with two elements it looks to see if the second element is a list. In both these paths it must always eventually check for the end-of-list notation BOOST_PP_NIL, which is an identifier in VMD. It is in this final test, that the end-of-list notation exists as a VMD identifier, where the UB could occur as explained in the topic 'Identifiers'.

## Distinguishing a seq and a tuple

As has previously been mentioned a single element tuple is also a one element seq.

However, as will be discussed later in the documentation, when VMD has to determine the type of such data, it always returns it as a tuple ( BOOST_VMD_TYPE_TUPLE ).

If our data consists of more than one consecutive tuple of a single element the data is a seq:

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

#define ST_DATA (somedata)(some_other_data)

BOOST_VMD_IS_SEQ(ST_DATA) will return 1
BOOST_VMD_IS_TUPLE(ST_DATA) will return 0
```

However if the data consists of a mixture we need to distinguish how VMD parses the data. The rule is that VMD always parses a single element tuple as a tuple unless it is followed by one or more single element tuples, in which case it is a seq.

```
#define ST_DATA (somedata)(element1,element2)
```

VMD parses the above data as 2 consecutive tuples. The first tuple is the single element tuple '(somedata)' and the second tuple is the multi element tuple '(element1,element2)'.

```
#define ST_DATA (element1,element2)(somedata)
```

VMD parses the above data as 2 consecutive tuples. The first tuple is the multi element tuple '(element1,element2)' and the second tuple is the single element tuple '(somedata)'.

```
#define ST_DATA (somedata)(some_other_data)(element1,element2)
```

VMD parses the above data as a seq followed by a tuple. The seq is '(somedata)(some_other_data)' and the tuple is '(element1,element2)'.

## Empty Boost PP data types

An array and a list can be empty.

An empty array has the form '(0,())', and is a perfectly valid array.

You can test for an empty array using the macro BOOST_VMD_IS_EMPTY_ARRAY.

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

#define AN_ARRAY (1,(1))
#define AN_EMPTY_ARRAY (0,())

BOOST_VMD_IS_ARRAY(AN_ARRAY) will return 1
BOOST_VMD_IS_ARRAY(AN_EMPTY_ARRAY) will return 1

BOOST_VMD_IS_EMPTY_ARRAY(AN_EMPTY_ARRAY) will return 1
BOOST_VMD_IS_EMPTY_ARRAY() will return 0
BOOST_VMD_IS_EMPTY_ARRAY(AN_ARRAY) will return 0
```

An empty list has the form 'BOOST_PP_NIL', and is a perfectly valid list.

You can test for an empty list using the macro BOOST_VMD_IS_EMPTY_LIST.

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

#define A_LIST (1,BOOST_PP_NIL)
#define AN_EMPTY_LIST BOOST_PP_NIL

BOOST_VMD_IS_LIST(A_LIST) will return 1
BOOST_VMD_IS_LIST(AN_EMPTY_LIST) will return 1

BOOST_VMD_IS_EMPTY_LIST(AN_EMPTY_LIST) will return 1
BOOST_VMD_IS_EMPTY_LIST() will return 0
BOOST_VMD_IS_EMPTY_LIST(A_LIST) will return 0
```

Neither seqs or tuples can be empty. Because of this if you convert from an empty array or list to a seq or tuple using Boost PP macros to do so you will get undefined behavior.

The syntax '()', which is called an empty parenthesis, is neither a zero-element seq or a tuple consisting of no elements. Rather it is either a one-element seq whose content is emptiness or a single-element tuple whose content is emptiness.

VMD supports the syntax of an empty parenthesis. You can test for it using the macro BOOST_VMD_IS_PARENS_EMPTY.

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

#define EMPTY_PARENS ()
#define TUPLE (0)
#define SEQ (0)(1)

BOOST_VMD_IS_TUPLE(EMPTY_PARENS) will return 1
BOOST_VMD_IS_SEQ(EMPTY_PARENS) will return 1

BOOST_VMD_IS_PARENS_EMPTY(EMPTY_PARENS) will return 1
BOOST_VMD_IS_PARENS_EMPTY() will return 0
BOOST_VMD_IS_PARENS_EMPTY(TUPLE) will return 0
BOOST_VMD_IS_PARENS_EMPTY(SEQ) will return 0
```

The VC++8 compiler ( Visual Studio 2005 ), which is the oldest VC++ version which VMD supports, has trouble working with the empty parenthesis syntax. Therefore if you have to use VC++8 avoid its use, otherwise you should be fine using it if you desire.

## Using a tuple instead of an array

When using variadic macros, the fact that an array can be empty is its only advantage over a tuple. Otherwise using a tuple is always easier since the syntax is simpler; you never have to notate the tuple's size.

Since VMD fully supports passing and returning emptiness you can use a tuple instead of an array in all situations and simply pass or return emptiness to represent an "empty" tuple, and as an equivalent to an empty array.

## Usage

You can use the general 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/is_array.hpp> //  for the BOOST_VMD_IS_ARRAY macro
#include <boost/vmd/is_list.hpp> //  for the BOOST_VMD_IS_LIST macro
#include <boost/vmd/is_seq.hpp> //  for the BOOST_VMD_IS_SEQ macro
#include <boost/vmd/is_tuple.hpp> // for the BOOST_VMD_IS_TUPLE macro.

#include <boost/vmd/is_empty_array.hpp> // for the BOOST_VMD_IS_EMPTY_ARRAY macro.
#include <boost/vmd/is_empty_list.hpp> // for the BOOST_VMD_IS_EMPTY_LIST macro.
#include <boost/vmd/is_parens_empty.hpp> // for the BOOST_VMD_IS_PARENS_EMPTY 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. The general name I will use in this documentation for these specific macros is "identifying macros." The identifying macros also share with BOOST_VMD_IS_EMPTY the inherent flaw mentioned when discussing BOOST_VMD_IS_EMPTY, 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.
- even if the function-like macro takes one parameter, passing emptiness to that macro could cause a preprocessing error.

The obvious way to avoid the BOOST_VMD_IS_EMPTY problem with 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 has 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, but 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.

# Generic macros for working with data types

Besides the specific macros for working with data types VMD has a number of generic macros for parsing sequences.

# Parsing sequences

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, with its own simpler functionality, also offers another alternative to representing data.

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 some entire input, to be part of a single parameter of input data as a series of data, as long as all the top-level data of such a single parameter is of some VMD data type. What this means is that if some input consists of a series of data types it is possible to extract the data for each data type in that series.

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

## Sequence definition

I will be calling such input data, which consists of all top-level data types in a series, by the term of a 'sequence'. Each separate data type in the sequence is called an 'element'. In this definition of a 'sequence' we can have 0 or more elements, so that a sequence is a general name for any VMD input. A sequence is therefore any input VMD can parse, whether it is emptiness, a single element, or more than one element in a series. Therefore when we speak of VMD macros parsing input data we are really speaking of VMD macros parsing a sequence. A sequence can therefore also be part of a Boost PP composite data type, or variadic data, and VMD can still parse such an embedded sequence if asked to do so.

## Sequence parsing

Parsing a sequence means that VMD can step through each element of a sequence sequentially, determine the type and data of each element, then move on to the next element. Parsing is sequential and can only be done in a forward direction, but it can be done any number of times. In C++ iterator terms parsing of a sequence is a forward iterator.

Working with a sequence is equivalent to using VMD macros generically.

Before I give an explanation of how to use a sequence using VMD generic functionality I would like to make two points:

- The possibility of working with a sequence which contains more than one data type for the preprocessor metaprogrammer 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 data types of a sequence takes more preprocessing time than functionality offered with Boost PP data types, because it is based on forward access through each top-level type of the sequence.

The one constraint in a sequence is that the top-level must consist of VMD data types, in other words preprocessor tokens which VMD understands. By top-level it is meant that a Boost PP composite data may have elements which VMD cannot parse but as long as the input consists of the composite data types and not the inner unparsable elements, VMD can parse the input. Therefore if pre-processor 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 data 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 any of the VMD identifying macros already discussed, 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 data types to do so. Such as:

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

or

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

or

```
identifier ( + ) number
```

or

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

The succeeding topics explain the VMD functionality for parsing a sequence for each individual VMD data type in that sequence.

## Sequence types

A VMD sequence can be seen as one of either three general types:

1. An empty sequence
2. A single element sequence
3. A multi-element sequence

An empty sequence is merely input that is empty, what VMD calls "emptiness". Use the previously explained BOOST_VMD_IS_EMPTY macro to test for an empty sequence.

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

#define AN_EMPTY_SEQUENCE

BOOST_VMD_IS_EMPTY(AN_EMPTY_SEQUENCE) will return 1
```

The type of an empty sequence is BOOST_VMD_TYPE_EMPTY.

A single element sequence is a single VMD data type. This is what we have been previously discussing as data which VMD can parse in this documentation with our identifying macros. You can use the BOOST_VMD_IS_UNARY macro to test for a single element sequence.

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

#define A_SINGLE_ELEMENT_SEQUENCE (1,2)

BOOST_VMD_IS_UNARY(A_SINGLE_ELEMENT_SEQUENCE) will return 1
```

The type of a single element sequence is the type of the individual data type. In our example above the type of A_SINGLE_ELEMENT_SEQUENCE is BOOST_VMD_TYPE_TUPLE.

A multi-element sequence consists of more than one data type. This is the "new" type which VMD can parse. You can use the BOOST_VMD_IS_MULTI macro to test for a multi-element sequence.

```
#define A_MULTI_ELEMENT_SEQUENCE (1,2) (1)(2) 45
```

The A_MULTI_ELEMENT_SEQUENCE consists of a tuple followed by a seq followed by a number.

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

BOOST_VMD_IS_MULTI(A_MULTI_ELEMENT_SEQUENCE) will return 1
```

The type of a multi-element sequence is always BOOST_VMD_TYPE_SEQUENCE.

The type of a sequence can be obtained generically with the BOOST_VMD_GET_TYPE macro. We will be explaining this further in the documentation.

### Sequence size

The size of any sequence can be accessed using the BOOST_VMD_SIZE macro. For an empty sequence the size is always 0. For a single element sequence the size is always 1. For a multi-element sequence the size is the number of individual top-level data types in the sequence.

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

BOOST_VMD_SIZE(AN_EMPTY_SEQUENCE) will return 0
BOOST_VMD_SIZE(A_SINGLE_ELEMENT_SEQUENCE) will return 1
BOOST_VMD_SIZE(A_MULTI_ELEMENT_SEQUENCE) will return 3
```

### Using VMD to parse sequence input

For a VMD sequence essentially two ways of parsing into individual data types are offered by the VMD library:

1. The sequence can be converted to any of the Boost PP data types, or to variadic data, where each individual data type in the sequence becomes a separate element of the particular composite data type chosen. The conversion to a particular Boost PP data type or variadic data is slow, because it is based on forward access through each top-level type of the sequence, but afterwards accessing any individual element is as fast as accessing any element in the Boost PP data type or among variadic data.

2. The sequence can be accessed directly through its individual elements. This is slower than accessing an element of a Boost PP data type or variadic data but offers conceptual access to the original sequence as a series of elements.

These two techniques will be discussed in succeeding topics.

## Converting sequences

The easiest way to work with a sequence is to convert it to a Boost PP data type. Likewise you can also convert a sequence to variadic data even though the Boost PP data types have much greater functionality.

To convert a sequence to a Boost PP data type or variadic data the macros to be used are:

- BOOST_VMD_TO_ARRAY(sequence) to convert the sequence to an array
- BOOST_VMD_TO_LIST(sequence) to convert the sequence to a list
- BOOST_VMD_TO_SEQ(sequence) to convert the sequence to a seq
- BOOST_VMD_TO_TUPLE(sequence) to convert the sequence to a tuple
- BOOST_VMD_ENUM(sequence) to convert the sequence to variadic data

After the conversion the elements of a sequence become the elements of the corresponding composite data type.

Once the elements of the sequence have been converted to the elements of the composite data type the full power of that composite data type can be used to process each element. Furthermore the programmer can use VMD to discover the type of an individual element for further processing.

For single element sequences the result is always a single element composite data type. For multi-element sequences the result is always a composite data type of more than one element.

For a sequence that is empty the result is emptiness when converting to a seq, tuple, or variadic data; the result is an empty array or list when converting to each of those composite data types respectively.

```
#include <boost/vmd/enum.hpp>
#include <boost/vmd/to_array.hpp>
#include <boost/vmd/to_list.hpp>
#include <boost/vmd/to_seq.hpp>
#include <boost/vmd/to_tuple.hpp>

#define BOOST_VMD_REGISTER_ANID (ANID)

#define SEQUENCE_EMPTY
#define SEQUENCE_SINGLE 35
#define SEQUENCE_SINGLE_2 ANID
#define SEQUENCE_MULTI (0,1) (2)(3)(4)
#define SEQUENCE_MULTI_2 BOOST_VMD_TYPE_SEQ (2,(5,6))

BOOST_VMD_TO_ARRAY(SEQUENCE_EMPTY) will return an empty array '(0,())'
BOOST_VMD_TO_LIST(SEQUENCE_SINGLE) will return a one-element list '(35,BOOST_PP_NIL)'
BOOST_VMD_TO_SEQ(SEQUENCE_SINGLE_2) will return a one-element seq '(ANID)'
BOOST_VMD_TO_TUPLE(SEQUENCE_MULTI) will return a multi-element tuple '((0,1),(2)(3)(4))'
BOOST_VMD_ENUM(SEQUENCE_MULTI_2) will return multi-element variad↵
ic data 'BOOST_VMD_TYPE_SEQ,(2,(5,6))'
```

### Usage

You can use the general 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/to_array.hpp> //  for the BOOST_VMD_TO_ARRAY macro
#include <boost/vmd/to_list.hpp> //  for the BOOST_VMD_TO_LIST macro
#include <boost/vmd/to_seq.hpp> //  for the BOOST_VMD_TO_SEQ macro
#include <boost/vmd/to_tuple.hpp> // for the BOOST_VMD_TO_TUPLE macro.
#include <boost/vmd/enum.hpp> // for the BOOST_VMD_ENUM macro.
```

## Accessing a sequence element

It is possible to access an individual element of a sequence. The macro to do this is called BOOST_VMD_ELEM. The macro takes two required parameters. The required parameters are the element number to access and the sequence, in that order. The element number is a 0-based number and its maximum value should be one less than the size of the sequence.

The BOOST_VMD_ELEM macro returns the actual sequence element. If the first required parameter is greater or equal to the size of the sequence the macro returns emptiness. Because of this using BOOST_VMD_ELEM on an empty sequence, whose size is 0, always returns emptiness.

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

#define BOOST_VMD_REGISTER_ANAME (ANAME)
#define A_SEQUENCE (1,2,3) 46 (list_data1,(list_data2,BOOST_PP_NIL)) BOOST_VMD_TYPE_SEQ ANAME
#define AN_EMPTY_SEQUENCE

BOOST_VMD_ELEM(0,A_SEQUENCE) will return (1,2,3)
BOOST_VMD_ELEM(1,A_SEQUENCE) will return 46
BOOST_VMD_ELEM(2,A_SEQUENCE) will return (list_data1,(list_data2,BOOST_PP_NIL))
BOOST_VMD_ELEM(3,A_SEQUENCE) will return BOOST_VMD_TYPE_SEQ
BOOST_VMD_ELEM(4,A_SEQUENCE) will return ANAME

BOOST_VMD_ELEM(5,A_SEQUENCE) will return emptiness
BOOST_VMD_ELEM(0,AN_EMPTY_SEQUENCE) will return emptiness
```

Accessing an element of a sequence directly is slower than accessing an element of a Boost PP data type or even variadic data, since each access has to directly cycle through each element of the sequence to get to the one being accessed. The process of sequentially parsing each element again each time is slower than accessing a Boost PP data type element.

### Usage

You can use the general header file:

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

or you can use the individual header file:

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

# Getting the type of data

VMD has the ability to retrieve the type of any data which it can parse, which means any VMD sequence. The macro to do this is called BOOST_VMD_GET_TYPE and it takes a single required parameter, which is a VMD sequence.

It returns one of the types previously discussed when introducing v-types as an identifier subset. As explained previously in that topic a v-type is fully recognized by VMD macros and can be part of a sequence and passed as VMD data just like all the other data types VMD recognizes.

When BOOST_VMD_GET_TYPE returns the type of data it returns by default the most specific type that the data can be. This means that non-empty lists and arrays are returned as such, not as tuples, and numbers and types and empty lists are returned as such, not as identifiers.

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

#define BOOST_VMD_REGISTER_ANID (ANID)
#define SEQUENCE_EMPTY
#define SEQUENCE_MULTI (1,2,3) 88
#define SEQUENCE1 (3,(1,2,3))
#define SEQUENCE2 ANID
#define SEQUENCE3 (1,(2,(3,BOOST_PP_NIL)))
#define SEQUENCE4 1
#define SEQUENCE5 (1)(2)(3)
#define SEQUENCE6 (1,2,3)
#define SEQUENCE7 BOOST_VMD_TYPE_NUMBER

BOOST_VMD_GET_TYPE(SEQUENCE_EMPTY) will return BOOST_VMD_TYPE_EMPTY
BOOST_VMD_GET_TYPE(SEQUENCE_MULTI) will return BOOST_VMD_TYPE_SEQUENCE
BOOST_VMD_GET_TYPE(SEQUENCE1) will return BOOST_VMD_TYPE_ARRAY
BOOST_VMD_GET_TYPE(SEQUENCE2) will return BOOST_VMD_TYPE_IDENTIFIER
BOOST_VMD_GET_TYPE(SEQUENCE3) will return BOOST_VMD_TYPE_LIST
BOOST_VMD_GET_TYPE(SEQUENCE4) will return BOOST_VMD_TYPE_NUMBER
BOOST_VMD_GET_TYPE(SEQUENCE5) will return BOOST_VMD_TYPE_SEQ
BOOST_VMD_GET_TYPE(SEQUENCE6) will return BOOST_VMD_TYPE_TUPLE
BOOST_VMD_GET_TYPE(SEQUENCE7) will return BOOST_VMD_TYPE_TYPE
```

### Usage

You can use the general header file:

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

or you can use the individual header file:

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

for the BOOST_VMD_GET_TYPE macro.

# Testing for equality and inequality

VMD allows the programmer to test generically for the equality or inequality of any value which VMD can parse. This includes emptiness, identifiers, numbers, types, arrays, lists, seqs, tuples, and multi-element sequences.

The macro to test for equality is called BOOST_VMD_EQUAL and it has two required parameters which are the two values against which to test. The values can be any VMD data type.

For the composite data types of array, list, seq, and tuple, or any of those types in a multi-element sequence, the elements of those types must also be a data type which VMD can parse. BOOST_VMD_EQUAL recursively parses the elements in a composite data type for equality, up to a level of 16 inner types, to test that one composite type equals another composite type. The requirement, that composite elements must also be a data type which VMD can parse, is different from most other macros in the VMD library, where only the top-level composite type need be parsed enough to determine the type of the data. If BOOST_VMD_EQUAL encounters a data type which it cannot parse the result will be UB.

VMD identifiers used in equality testing must be registered and pre-detected. All numbers and types are already registered/pre-detected for equality testing so it is only user-defined identifiers which must be registered and pre-detected. If an identifier has not been both registered and predetected it will never be equal to the same identifier value, so it will always fail equality testing, although it will not give a preprocessing error doing so.

The BOOST_VMD_EQUAL macro returns 1 if both parameters are equal and 0 if the parameters are not equal.

Conversely to test for inequality, of the same values as are required in testing for equality, the VMD library has the macro BOOST_VMD_NOT_EQUAL. This macro is simply a complement of the BOOST_VMD_EQUAL macro. If BOOST_VMD_EQUAL returns 1 then then BOOST_VMD_NOT_EQUAL returns 0 and if BOOST_VMD_EQUAL returns 0 then BOOST_VMD_NOT_EQUAL returns 1.

The BOOST_VMD_EQUAL and BOOST_VMD_NOT_EQUAL macros are called "equality macros".

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

#define BOOST_VMD_REGISTER_AN_ID1 (AN_ID1)
#define BOOST_VMD_REGISTER_AN_ID2 (AN_ID2)

#define BOOST_VMD_DETECT_AN_ID1_AN_ID1
#define BOOST_VMD_DETECT_AN_ID2_AN_ID2

#define AN_IDENTIFIER1 AN_ID1
#define AN_IDENTIFIER2 AN_ID2
#define AN_IDENTIFIER3 AN_ID1 // same as AN_IDENTIFIER1 = AN_ID1

#define A_NUMBER1 33
#define A_NUMBER2 145
#define A_NUMBER3 33 // same as A_NUMBER1 = 33

#define A_TUPLE1 (AN_IDENTIFIER1,A_NUMBER1)
#define A_TUPLE2 (AN_IDENTIFIER1,A_NUMBER2)
#define A_TUPLE3 (AN_IDENTIFIER3,A_NUMBER3) // same as A_TUPLE1 = (AN_ID1,33)

#define A_SEQ1 (A_NUMBER1)(A_TUPLE1)
#define A_SEQ2 (A_NUMBER2)(A_TUPLE2)
#define A_SEQ3 (A_NUMBER3)(A_TUPLE3) // same as A_SEQ1 = (33)((AN_ID1,33))

BOOST_VMD_EQUAL(AN_IDENTIFIER1,AN_IDENTIFIER2) will return 0
BOOST_VMD_EQUAL(AN_IDENTIFIER1,AN_IDENTIFIER3) will return 1

BOOST_VMD_EQUAL(A_NUMBER1,A_NUMBER2) will return 0
BOOST_VMD_EQUAL(A_NUMBER1,A_NUMBER3) will return 1

BOOST_VMD_EQUAL(A_TUPLE1,A_TUPLE2) will return 0
BOOST_VMD_EQUAL(A_TUPLE1,A_TUPLE3) will return 1

BOOST_VMD_EQUAL(A_SEQ1,A_SEQ2) will return 0
BOOST_VMD_EQUAL(A_SEQ1,A_SEQ3) will return 1
```

When BOOST_VMD_EQUAL tests for equality it always parses data for their most specific types. The reason for this is that a valid tuple, which is an invalid list or array, can never be compared completely because all elements of that tuple are not data types which VMD can parse. Therefore VMD always tests equality based on the most specific type for any value being tested, which speeds up testing for the more specific tuple data types such as lists and arrays.

```
#define TUPLE_IS_ARRAY1 (2,(3,4))
#define TUPLE_IS_ARRAY2 (2,(4,5))
#define TUPLE_IS_ARRAY3 (2,(3,4))

#define TUPLE_IS_LIST1 (55,BOOST_PP_NIL)
#define TUPLE_IS_LIST2 (135,BOOST_PP_NIL)
#define TUPLE_IS_LIST3 (55,BOOST_PP_NIL)

#define TUPLE_IS_LIST_OR_ARRAY1 (2,(3,BOOST_PP_NIL))
#define TUPLE_IS_LIST_OR_ARRAY2 (2,(4,BOOST_PP_NIL))
#define TUPLE_IS_LIST_OR_ARRAY3 (2,(3,BOOST_PP_NIL))

#define TUPLE_BUT_INVALID_ARRAY1 (&2,(3,4))
#define TUPLE_BUT_INVALID_ARRAY2 (&2,(4,4))
#define TUPLE_BUT_INVALID_ARRAY3 (&2,(3,4))

#define TUPLE_BUT_INVALID_LIST1 (55,^BOOST_PP_NIL)
#define TUPLE_BUT_INVALID_LIST2 (135,^BOOST_PP_NIL)
#define TUPLE_BUT_INVALID_LIST3 (55,^BOOST_PP_NIL)
```

All of the constructs above are valid tuples.

The first three are valid arrays, so they will be parsed and compared as arrays, so that they can be used as in:

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

BOOST_VMD_EQUAL(TUPLE_IS_ARRAY1,TUPLE_IS_ARRAY2) will return 0
BOOST_VMD_EQUAL(TUPLE_IS_ARRAY1,TUPLE_IS_ARRAY3) will return 1
```

The next three are valid lists, so they will be parsed and compared as lists, so that they can be used as in:

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

BOOST_VMD_EQUAL(TUPLE_IS_LIST1,TUPLE_IS_LIST2) will return 0
BOOST_VMD_EQUAL(TUPLE_IS_LIST1,TUPLE_IS_LIST3) will return 1
```

The next three are valid lists or arrays but will be parsed as lists because lists are more specific than arrays. They can be used as in:

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

BOOST_VMD_EQUAL(TUPLE_IS_LIST_OR_ARRAY1,TUPLE_IS_LIST_OR_ARRAY2) will return 0
BOOST_VMD_EQUAL(TUPLE_IS_LIST_OR_ARRAY1,TUPLE_IS_LIST_OR_ARRAY3) will return 1
```

The next three are valid tuples but invalid arrays. The BOOST_VMD_EQUAL macro attempts to parse them as the most specific type they can be, which is an array. But the attempt to parse them as arrays will lead to UB because the number which signifies the size of the array is invalid as a number. Now let us suppose we should parse them as the less specific type of a tuple instead of as an array. This will still give UB if we will attempt to compare the first tuple element against a corresponding first tuple element of another tuple, and when we do will again encounter UB because it is not a data type VMD can parse.

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

BOOST_VMD_EQUAL(TUPLE_BUT_INVALID_ARRAY1,TUPLE_BUT_INVALID_ARRAY1) will generate UB
BOOST_VMD_EQUAL(TUPLE_BUT_INVALID_ARRAY1,TUPLE_BUT_INVALID_ARRAY1) will generate UB
```

The next three are valid tuples but invalid lists. The BOOST_VMD_EQUAL macro attempts to parse them as the most specific type they can be, which is a list. But the attempt to parse them as lists will lead to UB because the identifier which signifies the end-of-list is invalid as an identifier. Now let us suppose we should parse them as the less specific type of a tuple instead of as a list. This will still give UB if we will attempt to compare the second tuple element against a corresponding second tuple element of another tuple, and when we do will again encounter UB because it is not a data type VMD can parse.

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

BOOST_VMD_EQUAL(TUPLE_BUT_INVALID_LIST1,TUPLE_BUT_INVALID_LIST2) will generate UB
BOOST_VMD_EQUAL(TUPLE_BUT_INVALID_LIST1,TUPLE_BUT_INVALID_LIST3) will generate UB
```

It is possible that a composite data type which has an element which VMD cannot parse will not give UB when compared for equality, but rather just the test for equality will fail. This can occur if the algorithm which tests for equality tests false before parsing of the particular element. Such a situation might be:

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

#define A_TUPLE1 (3,4,"astring")
#define A_TUPLE2 (3,4)

BOOST_VMD_EQUAL(A_TUPLE1,A_TUPLE2) will return 0 rather than generate UB
```

The reason the above correctly returns 0, rather than generate UB when VMD attempts to parse '"astring"', which is not a data type VMD can parse, is because the algorithm for testing equality tests whether or not the tuples have the same number of elements before

it tests for the equality of each element. This is just one example where testing for equality may fail before UB is generated when BOOST_VMD_EQUAL attempts to parse a data type which it cannot handle. Nevertheless the general rule should still be considered that for BOOST_VMD_EQUAL/BOOT_VMD_NOT_EQUAL all data types, even an element of a composite data type, must be a VMD data type if the macro is to work properly, else UB could occur.

## Usage

You can use the general header file:

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

or you can use the individual header files:

```
#include <boost/vmd/equal.hpp> for the BOOST_VMD_EQUAL macro
#include <boost/vmd/not_equal.hpp> for the BOOST_VMD_NOT_EQUAL macro
```

# Macros with modifiers

The basic functionality for VMD macros parsing data types has been given using the required parameters of those macros. This basic functionality may be perfectly adequate for macro programmers to use VMD effectively in their programming efforts.

A number of those macros take optional parameters, called in general "modifiers", which enhance or change the functionality of those macros in various ways. All modifiers are VMD identifiers.

Some modifiers are identifiers, both registered and pre-detected, starting with BOOST_VMD_. These are termed VMD "specific modifiers" and change the expansion of a macro in various ways.

Modifiers may also be user-defined identifiers in some situations which will be subsequently explained. These user-defined identifiers are termed "user-defined modifiers".

In all situations modifiers are optional parameters which are parsed by VMD to provide enhanced functionality for some of its macros. They are never required as part of the basic functionality of a macro.

When modifiers are used as optional arguments to a macro they can be input after the required parameters in any order and VMD will still handle the optional parameters correctly.

For any particular macro if a specific modifier is not appropriate it is just ignored. This means that VMD never generates a preprocessing error or gives an incorrect result just because a specific modifier does not apply for a particular macro. Any modifier which is not recognized as a specific modifier is treated as a user-defined modifier. In cases where a user-defined modifier is not appropriate it is also just ignored.

The situations where modifiers can be used to enhance the basic functionality of VMD macros can be divided by particular types of specific modifiers. Each particular type of a specific modifier has a particular name given to it in this documentation, functionality, set of identifiers, and may be used as optional parameters in one or more designated macros depending on the specific modifier type.

When more than one specific modifier from a particular type of modifier is specified as an optional parameter the last specified takes effect. This allows the programmer to override a specific modifier by adding the overridden identifier as an optional argument to the end of the macro's invocation.

Header files for specific modifiers are automatically included when the header files for macros taking those specific modifiers are included.

Header files for user-defined modifiers, which register and pre-detect those user-defined modifiers, must be included as needed by the programmer using those modifiers.

The following topics will explain each particular type of modifier and where it may be used.

# Return type modifiers

A number of macros are capable of returning the type of data as a v-type rather than, or along with, the data itself. The most obvious of these is BOOST_VMD_GET_TYPE which generically returns the type of the input.

Return type modifiers turn on, turn off, or change the type of the data returned in some way.

These modifiers are:

- BOOST_VMD_RETURN_NO_TYPE, do not return the type of data.
- BOOST_VMD_RETURN_TYPE, return the type of data parsing any tuple-like syntactical construct as its most specific type. This means that any tuple-like construct is parsed first as a possible list, next as a possible array if it is not a list, and finally as a tuple if it is not a list or an array.
- BOOST_VMD_RETURN_TYPE_LIST, parse any tuple-like syntactical construct first as a possible list and only then as a tuple if it is not a list.

- BOOST_VMD_RETURN_TYPE_ARRAY, parse any tuple-like syntactical construct first as a possible array and only then as a tuple if it is not an array.
- BOOST_VMD_RETURN_TYPE_TUPLE, parse any tuple-like syntactical construct only as a tuple.

When VMD parses input generically it must determine the type of each data element of the input. For nearly all of the VMD data types this is never a problem. For the array or list data types this can be a problem, as explained when discussing parsing arrays and lists respectively using the specific macros BOOST_VMD_IS_ARRAY and BOOST_VMD_IS_LIST. The problem is that a valid tuple can be an invalid list or an invalid array, whose parsing as the more specific type will lead to UB. Because of this when VMD parses input generically, and only the data of an element is needed to continue parsing correctly, it parses all tuple-like data as a tuple and never as a list or an array.

When VMD parses input generically, and the type of the data is required in some way as part of the return of a macro, VMD by default parses for the most specific type of each data element in order to return the most accurate type. In this situation by default the BOOST_VMD_RETURN_TYPE modifier is internally in effect without having to be specified.

If more than one of the return type modifiers are specified as optional parameters last one specified is in effect.

## Usage with BOOST_VMD_GET_TYPE

The only macro in which VMD without the use of modifiers is being asked to return the type of data is BOOST_VMD_GET_TYPE. For this macro the BOOST_VMD_RETURN_TYPE modifier is internally in effect so if no return type modifiers are input as optional parameters BOOST_VMD_GET_TYPE looks for the most specific type.

For the BOOST_VMD_GET_TYPE macro the optional return type modifier BOOST_VMD_RETURN_NO_TYPE, if specified, is always ignored since the purpose of BOOST_VMD_GET_TYPE is solely to return the v-type.

Let's look at how this works with BOOST_VMD_GET_TYPE by specifying VMD sequences that have tuples which may or may not be valid lists or arrays.

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

#define TUPLE_IS_ARRAY (2,(3,4))
#define TUPLE_IS_LIST (anydata,BOOST_PP_NIL)
#define TUPLE_IS_LIST_OR_ARRAY (2,(3,BOOST_PP_NIL))
#define TUPLE_BUT_INVALID_ARRAY (&2,(3,4))
#define TUPLE_BUT_INVALID_LIST (anydata,^BOOST_PP_NIL)
#define SEQUENCE_EMPTY
#define SEQUENCE_MULTI TUPLE_BUT_INVALID_ARRAY TUPLE_BUT_INVALID_LIST

BOOST_VMD_GET_TYPE(TUPLE_IS_ARRAY) will return BOOST_VMD_TYPE_ARRAY, the most specific type
BOOST_VMD_GET_TYPE(TUPLE_IS_ARRAY,BOOST_VMD_RETURN_TYPE_TUPLE) will return BOOST_VMD_TYPE_TUPLE
BOOST_VMD_GET_TYPE(TUPLE_IS_ARRAY,BOOST_VMD_RETURN_TYPE_ARRAY) will return BOOST_VMD_TYPE_ARRAY
BOOST_VMD_GET_TYPE(TUPLE_IS_ARRAY,BOOST_VMD_RETURN_TYPE_LIST) will return BOOST_VMD_TYPE_TUPLE

BOOST_VMD_GET_TYPE(TUPLE_IS_LIST) will return BOOST_VMD_TYPE_LIST, the most specific type
BOOST_VMD_GET_TYPE(TUPLE_IS_LIST,BOOST_VMD_RETURN_TYPE_TUPLE) will return BOOST_VMD_TYPE_TUPLE
BOOST_VMD_GET_TYPE(TUPLE_IS_LIST,BOOST_VMD_RETURN_TYPE_ARRAY) will return BOOST_VMD_TYPE_TUPLE
BOOST_VMD_GET_TYPE(TUPLE_IS_LIST,BOOST_VMD_RETURN_TYPE_LIST) will return BOOST_VMD_TYPE_LIST

BOOST_VMD_GET_TYPE(TUPLE_IS_LIST_OR_ARRAY) will return BOOST_VMD_TYPE_LIST, the most specific type
BOOST_VMD_GET_TYPE(TUPLE_IS_LIST_OR_ARRAY,BOOST_VMD_RETURN_TYPE_TUPLE) will re↵
turn BOOST_VMD_TYPE_TUPLE
BOOST_VMD_GET_TYPE(TUPLE_IS_LIST_OR_ARRAY,BOOST_VMD_RETURN_TYPE_ARRAY) will re↵
turn BOOST_VMD_TYPE_ARRAY
BOOST_VMD_GET_TYPE(TUPLE_IS_LIST_OR_ARRAY,BOOST_VMD_RETURN_TYPE_LIST) will re↵
turn BOOST_VMD_TYPE_LIST

BOOST_VMD_GET_TYPE(TUPLE_BUT_INVALID_ARRAY) will give UB
BOOST_VMD_GET_TYPE(TUPLE_BUT_INVALID_ARRAY,BOOST_VMD_RETURN_TYPE_TUPLE) will re↵
turn BOOST_VMD_TYPE_TUPLE
BOOST_VMD_GET_TYPE(TUPLE_BUT_INVALID_ARRAY,BOOST_VMD_RETURN_TYPE_ARRAY) will give UB
BOOST_VMD_GET_TYPE(TUPLE_BUT_INVALID_ARRAY,BOOST_VMD_RETURN_TYPE_LIST) will re↵
turn BOOST_VMD_TYPE_TUPLE

BOOST_VMD_GET_TYPE(TUPLE_BUT_INVALID_LIST) will give UB
BOOST_VMD_GET_TYPE(TUPLE_BUT_INVALID_LIST,BOOST_VMD_RETURN_TYPE_TUPLE) will re↵
turn BOOST_VMD_TYPE_TUPLE
BOOST_VMD_GET_TYPE(TUPLE_BUT_INVALID_LIST,BOOST_VMD_RETURN_TYPE_ARRAY) will re↵
turn BOOST_VMD_TYPE_TUPLE
BOOST_VMD_GET_TYPE(TUPLE_BUT_INVALID_LIST,BOOST_VMD_RETURN_TYPE_LIST) will give UB

BOOST_VMD_GET_TYPE(SEQUENCE_EMPTY)
   will always return BOOST_VMD_TYPE_EMPTY even if we add any return type modifiers
BOOST_VMD_GET_TYPE(SEQUENCE_MULTI)
   will always return BOOST_VMD_TYPE_SEQUENCE even if we add any return type modifiers
```

## Usage with sequence converting macros

The sequence converting macros converts a sequence to a composite Boost PP data type or to variadic data, where each element's data in the sequence becomes an element in the destination composite type. The macros are:

- BOOST_VMD_TO_ARRAY, converts the sequence to an array
- BOOST_VMD_TO_LIST, converts the sequence to a list
- BOOST_VMD_TO_SEQ, converts the sequence to a seq
- BOOST_VMD_TO_TUPLE, converts the sequence to a tuple
- BOOST_VMD_ENUM, converts the sequence to variadic data

When it does the conversion, using just the required parameter of the sequence itself, it converts only the data value of each sequence element to the elements of a composite Boost PP data type or variadic data. Because it needs only the data value of each sequence element it determines the type of each sequence element as the the most general type that it can be. This means that all tuple-like data are parsed as tuples rather than as possible lists or arrays.

Using a return type modifier we can convert from a VMD sequence to a Boost PP composite data type or variadic data and retain the type of data of each element in the sequence as part of the conversion. When doing this each of the converted elements of the composite data type becomes a two-element tuple where the first element is the type of the data and the second element is the data itself.

For the sequence conversion macros the default return type modifier internally set is BOOST_VMD_RETURN_NO_TYPE, which means that the type is not retained. By specifying another optional return type modifier we tell the conversion to preserve the type in the conversion output.

If the sequence is empty, since there are no sequence elements, any return type modifier we use accomplishes nothing but is fine to use.

First we show how sequence conversion macros work with the BOOST_VMD_RETURN_TYPE modifier, which always parses for the most specific type.

```
#include <boost/vmd/enum.hpp>
#include <boost/vmd/to_array.hpp>
#include <boost/vmd/to_list.hpp>
#include <boost/vmd/to_seq.hpp>
#include <boost/vmd/to_tuple.hpp>

#define BOOST_VMD_REGISTER_ANID (ANID)
#define SEQUENCE_EMPTY_1
#define SEQUENCE_SINGLE 35
#define SEQUENCE_SINGLE_ID ANID
#define SEQUENCE_SINGLE_ARRAY (3,(0,1,2))
#define SEQUENCE_SINGLE_LIST (data,(more_data,BOOST_PP_NIL))
#define SEQUENCE_MULTI_1 (0,1) (2)(3)(4)
#define SEQUENCE_MULTI_2 BOOST_VMD_TYPE_SEQ (2,(5,6))

BOOST_VMD_TO_ARRAY(SEQUENCE_EMPTY_1) will return an empty array '(0,())'
BOOST_VMD_TO_ARRAY(SEQUENCE_EMPTY_1,BOOST_VMD_RETURN_TYPE) will return an empty array '(0,())'

BOOST_VMD_TO_LIST(SEQUENCE_SINGLE) will return a one-element list '(35,BOOST_PP_NIL)'
BOOST_VMD_TO_LIST(SEQUENCE_SINGLE,BOOST_VMD_RETURN_TYPE)
   will return a one-element list '((BOOST_VMD_TYPE_NUMBER,35),BOOST_PP_NIL)'

BOOST_VMD_TO_SEQ(SEQUENCE_SINGLE_ID) will return a one-element seq '(ANID)'
BOOST_VMD_TO_SEQ(SEQUENCE_SINGLE_ID,BOOST_VMD_RETURN_TYPE)
   will return a one-element seq '((BOOST_VMD_TYPE_IDENTIFIER,ANID))'

BOOST_VMD_TO_TUPLE(SEQUENCE_SINGLE_ARRAY) will return a single element tuple '((3,(0,1,2)))'
BOOST_VMD_TO_TUPLE(SEQUENCE_SINGLE_ARRAY,BOOST_VMD_RETURN_TYPE)
   will return a single element tuple '((BOOST_VMD_TYPE_ARRAY,(3,(0,1,2))))'

BOOST_VMD_ENUM(SEQUENCE_SINGLE_LIST) will return the single-ele↵
ment '(data,(more_data,BOOST_PP_NIL))'
BOOST_VMD_ENUM(SEQUENCE_SINGLE_LIST,BOOST_VMD_RETURN_TYPE)
   will return the single element '(BOOST_VMD_TYPE_LIST,(data,(more_data,BOOST_PP_NIL)))'

BOOST_VMD_TO_TUPLE(SEQUENCE_MULTI_1) will return a multi-element tuple '((0,1),(2)(3)(4))'
BOOST_VMD_TO_TUPLE(SEQUENCE_MULTI_1,BOOST_VMD_RETURN_TYPE)
   will return a multi-ele↵
ment tuple '((BOOST_VMD_TYPE_TUPLE,(0,1)),(BOOST_VMD_TYPE_SEQ,(2)(3)(4)))'

BOOST_VMD_ENUM(SEQUENCE_MULTI_2) will return multi-element variad↵
ic data 'BOOST_VMD_TYPE_SEQ,(2,(5,6))'
BOOST_VMD_ENUM(SEQUENCE_MULTI_2,BOOST_VMD_RETURN_TYPE)
   will return multi-element variad↵
ic data '(BOOST_VMD_TYPE_TYPE,BOOST_VMD_TYPE_SEQ),(BOOST_VMD_TYPE_ARRAY,(2,(5,6)))'
```

Lets look at how we might use other return type modifiers when doing conversions to avoid UB if we want the type as part of the conversion but the type might be a valid tuple while being an invalid list or array.

```
#define TUPLE_IS_VALID_ARRAY (2,(3,4))
#define TUPLE_IS_VALID_LIST (anydata,BOOST_PP_NIL)
#define TUPLE_BUT_INVALID_ARRAY_2 (&2,(3,4))
#define TUPLE_BUT_INVALID_LIST_2 (anydata,^BOOST_PP_NIL)

#define SEQUENCE_MULTI_T1 TUPLE_IS_VALID_ARRAY TUPLE_IS_VALID_LIST
#define SEQUENCE_MULTI_T2 TUPLE_BUT_INVALID_ARRAY_2 TUPLE_IS_VALID_LIST
#define SEQUENCE_MULTI_T3 TUPLE_IS_VALID_ARRAY TUPLE_BUT_INVALID_LIST_2
#define SEQUENCE_MULTI_T4 TUPLE_BUT_INVALID_ARRAY_2 TUPLE_BUT_INVALID_LIST_2
```

We present a number of uses of various sequence conversions with each of our four different sequences, and show their results.

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

BOOST_VMD_TO_SEQ(SEQUENCE_MULTI_T1,BOOST_VMD_RETURN_TYPE)
   will return the seq '((BOOST_VMD_TYPE_ARRAY,(2,(3,4)))) ((BOOST_VMD_TYPE_LIST,(any↵
data,BOOST_PP_NIL)))'
BOOST_VMD_TO_SEQ(SEQUENCE_MULTI_T1,BOOST_VMD_RETURN_TYPE_ARRAY)
   will return the seq '((BOOST_VMD_TYPE_ARRAY,(2,(3,4)))) ((BOOST_VMD_TYPE_TUPLE,(any↵
data,BOOST_PP_NIL)))'
BOOST_VMD_TO_SEQ(SEQUENCE_MULTI_T1,BOOST_VMD_RETURN_TYPE_LIST)
   will return the seq '((BOOST_VMD_TYPE_TUPLE,(2,(3,4)))) ((BOOST_VMD_TYPE_LIST,(any↵
data,BOOST_PP_NIL)))'
BOOST_VMD_TO_SEQ(SEQUENCE_MULTI_T1,BOOST_VMD_RETURN_TYPE_TUPLE)
   will return the seq '((BOOST_VMD_TYPE_TUPLE,(2,(3,4)))) ((BOOST_VMD_TYPE_TUPLE,(any↵
data,BOOST_PP_NIL)))'
```

The SEQUENCE_MULTI_T1 is a valid array followed by a valid list. All return type modifiers produce their results without any UBs.

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

BOOST_VMD_TO_TUPLE(SEQUENCE_MULTI_T2,BOOST_VMD_RETURN_TYPE)
   will produce UB when attempting to parse the invalid array as an array
BOOST_VMD_TO_TUPLE(SEQUENCE_MULTI_T2,BOOST_VMD_RETURN_TYPE_ARRAY)
   will produce UB when attempting to parse the invalid array as an array
BOOST_VMD_TO_TUPLE(SEQUENCE_MULTI_T2,BOOST_VMD_RETURN_TYPE_LIST)
   will return the tuple '((BOOST_VMD_TYPE_TUPLE,(&2,(3,4))),(BOOST_VMD_TYPE_LIST,(any↵
data,BOOST_PP_NIL)))'
BOOST_VMD_TO_TUPLE(SEQUENCE_MULTI_T2,BOOST_VMD_RETURN_TYPE_TUPLE)
   will return the tuple '((BOOST_VMD_TYPE_TUPLE,(&2,(3,4))),(BOOST_VMD_TYPE_TUPLE,(any↵
data,BOOST_PP_NIL)))'
```

The SEQUENCE_MULTI_T2 is an invalid array, but valid tuple, followed by a valid list.

In sequence conversion we will get UB whenever we use a return type modifier that parses the data type of the invalid array as an array. But if we use the return type modifiers BOOST_VMD_RETURN_TYPE_LIST or BOOST_VMD_RETURN_TYPE_TUPLE we are never parsing the invalid array as an array but as a tuple instead and therefore we successfully return the type of the invalid array as a BOOST_VMD_TYPE_TUPLE.

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

BOOST_VMD_TO_ARRAY(SEQUENCE_MULTI_T3,BOOST_VMD_RETURN_TYPE)
   will produce UB when attempting to parse the invalid list as a list
BOOST_VMD_TO_ARRAY(SEQUENCE_MULTI_T3,BOOST_VMD_RETURN_TYPE_LIST)
   will produce UB when attempting to parse the invalid list as a list
BOOST_VMD_TO_ARRAY(SEQUENCE_MULTI_T3,BOOST_VMD_RETURN_TYPE_ARRAY)
   will return the array '(2,((BOOST_VMD_TYPE_ARRAY,(2,(3,4))),(BOOST_VMD_TYPE_TUPLE,(any↵
data,^BOOST_PP_NIL))))'
BOOST_VMD_TO_ARRAY(SEQUENCE_MULTI_T3,BOOST_VMD_RETURN_TYPE_TUPLE)
   will return the array '(2,((BOOST_VMD_TYPE_TUPLE,(2,(3,4))),(BOOST_VMD_TYPE_TUPLE,(any↵
data,^BOOST_PP_NIL))))'
```

The SEQUENCE_MULTI_T3 is a valid array followed by an invalid list, but a valid tuple.

In sequence conversion we will get UBs whenever we use a return type modifier that parses the data type of the invalid list as a list. But if we use the return type modifiers BOOST_VMD_RETURN_TYPE_ARRAY or BOOST_VMD_RETURN_TYPE_TUPLE we are never parsing the invalid list as a list but as a tuple instead and therefore we successfully return the type of the invalid list as a BOOST_VMD_TYPE_TUPLE.

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

BOOST_VMD_TO_LIST(SEQUENCE_MULTI_T4,BOOST_VMD_RETURN_TYPE)
   will produce UB when attempting to parse the invalid array or invalid list
BOOST_VMD_TO_LIST(SEQUENCE_MULTI_T4,BOOST_VMD_RETURN_TYPE_ARRAY)
   will produce UB when attempting to parse the invalid array
BOOST_VMD_TO_LIST(SEQUENCE_MULTI_T4,BOOST_VMD_RETURN_TYPE_LIST)
   will produce UB when attempting to parse the invalid list
BOOST_VMD_TO_LIST(SEQUENCE_MULTI_T4,BOOST_VMD_RETURN_TYPE_TUPLE)
   will return the list '((BOOST_VMD_TYPE_TUPLE,(&2,(3,4))),((BOOST_VMD_TYPE_TUPLE,(any↵
data,^BOOST_PP_NIL)),BOOST_PP_NIL))'
```

The SEQUENCE_MULTI_T4 is an invalid array, but valid tuple, followed by a invalid list, but valid tuple.

In sequence conversion we will get UBs whenever we use a return type modifier that parses the sequence other than looking for only valid tuples. So here we must use the return type modifier BOOST_VMD_RETURN_TYPE_TUPLE for a sequence conversion without generating UBs.

## Usage with BOOST_VMD_ELEM

The default functionality of BOOST_VMD_ELEM when the required parameters are used is to return the particular element's data. When BOOST_VMD_ELEM does this it parses all elements of the sequence by determining the most general type of data for each element. The parsing algorithm stops when it reaches the element number whose data is returned. This means that all tuple-like data are parsed as tuples rather than as possible lists or arrays.

Using return type modifiers as optional parameters we can tell BOOST_VMD_ELEM to return the type of the element found, as well as its data, in the form of a two element tuple. The first element of the tuple is the type of the data and the second element of the tuple is the data itself.

When we use a return type modifier with BOOST_VMD_ELEM, which tells us to return the type of the data along with the data, the particular modifier only tells BOOST_VMD_ELEM how to parse the type of data for the element found. BOOST_VMD_ELEM will continue to parse elements in the sequence preceding the element found by determining the most general type of the data since this is the safest way of parsing the data itself.

Using the return type modifier BOOST_VMD_RETURN_TYPE with BOOST_VMD_ELEM is perfectly safe as long as the particular element found is not an invalid list or array, but a valid tuple. It is when the element found may be an invalid list or invalid array that we must use other return type modifiers in order to parse the type of the element correctly.

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

#define BOOST_VMD_REGISTER_ANID_E (ANID_E)
#define SEQUENCE_SINGLE_E 35
#define SEQUENCE_SINGLE_E2 ANID_E
#define SEQUENCE_MULTI_E (0,1) (2)(3)(4)
#define SEQUENCE_MULTI_E_2 BOOST_VMD_TYPE_SEQ (2,(5,6))

BOOST_VMD_ELEM(0,SEQUENCE_SINGLE_E) will return '35'
BOOST_VMD_ELEM(0,SEQUENCE_SINGLE_E,BOOST_VMD_RETURN_TYPE) will return '(BOOST_VMD_TYPE_NUMBER,35)'

BOOST_VMD_ELEM(0,SEQUENCE_SINGLE_E2) will return 'ANID_E'
BOOST_VMD_ELEM(0,SEQUENCE_SINGLE_E2,BOOST_VMD_RETURN_TYPE) will return '(BOOST_VMD_TYPE_IDENTI↵
FIER,ANID_E)'

BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E) will return '(2)(3)(4)'
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E,BOOST_VMD_RETURN_TYPE) will re↵
turn '(BOOST_VMD_TYPE_SEQ,(2)(3)(4))'

BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E_2) will return 'BOOST_VMD_TYPE_SEQ'
BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E_2,BOOST_VMD_RETURN_TYPE) will re↵
turn '(BOOST_VMD_TYPE_TYPE,BOOST_VMD_TYPE_SEQ)'
```

When we use the other return type modifiers with BOOST_VMD_ELEM we do so because the element we want may be an invalid list or an invalid array but a valid tuple and yet we still want its type returned as part of the result.

Let's look at how this works with BOOST_VMD_ELEM by specifying VMD sequences that have tuples which are in the form of arrays or lists which cannot be parsed as such by VMD without generating UBs.

```
#define TUPLE_IS_VALID_ARRAY_E (2,(3,4))
#define TUPLE_IS_VALID_LIST_E (anydata,BOOST_PP_NIL)
#define TUPLE_BUT_INVALID_ARRAY_E (&2,(3,4))
#define TUPLE_BUT_INVALID_LIST_E (anydata,^BOOST_PP_NIL)

#define SEQUENCE_MULTI_E1 TUPLE_IS_VALID_ARRAY_E TUPLE_IS_VALID_LIST_E
#define SEQUENCE_MULTI_E2 TUPLE_BUT_INVALID_ARRAY_E TUPLE_IS_VALID_LIST_E
#define SEQUENCE_MULTI_E3 TUPLE_IS_VALID_ARRAY_E TUPLE_BUT_INVALID_LIST_E
#define SEQUENCE_MULTI_E4 TUPLE_BUT_INVALID_ARRAY_E TUPLE_BUT_INVALID_LIST_E
```

We present a number of uses of BOOST_VMD_ELEM with each of our four different sequences, and show their results.

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

BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E1,BOOST_VMD_RETURN_TYPE) will return '(BOOST_VMD_TYPE_AR↵
RAY,(2,(3,4)))'
BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E1,BOOST_VMD_RETURN_TYPE_ARRAY) will return '(BOOST_VMD_TYPE_AR↵
RAY,(2,(3,4)))'
BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E1,BOOST_VMD_RETURN_TYPE_LIST) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(2,(3,4)))'
BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E1,BOOST_VMD_RETURN_TYPE_TUPLE) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(2,(3,4)))'
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E1,BOOST_VMD_RETURN_TYPE) will return '(BOOST_VMD_TYPE_LIST,(any↵
data,BOOST_PP_NIL))'
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E1,BOOST_VMD_RETURN_TYPE_ARRAY) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(anydata,BOOST_PP_NIL))'
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E1,BOOST_VMD_RETURN_TYPE_LIST) will re↵
turn '(BOOST_VMD_TYPE_LIST,(anydata,BOOST_PP_NIL))'
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E1,BOOST_VMD_RETURN_TYPE_TUPLE) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(anydata,BOOST_PP_NIL))'
```

The SEQUENCE_MULTI_E1 is a valid array followed by a valid list. All return type modifiers produce their results without any UBs.

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

BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E2,BOOST_VMD_RETURN_TYPE)
   will produce UB when attempting to parse the invalid array as an array
BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E2,BOOST_VMD_RETURN_TYPE_ARRAY)
   will produce UB when attempting to parse the invalid array as an array
BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E2,BOOST_VMD_RETURN_TYPE_LIST) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(&2,(3,4)))'
BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E2,BOOST_VMD_RETURN_TYPE_TUPLE) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(&2,(3,4)))'
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E2,BOOST_VMD_RETURN_TYPE) will return '(BOOST_VMD_TYPE_LIST,(any↵
data,BOOST_PP_NIL))'
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E2,BOOST_VMD_RETURN_TYPE_ARRAY) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(anydata,BOOST_PP_NIL))'
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E2,BOOST_VMD_RETURN_TYPE_LIST) will re↵
turn '(BOOST_VMD_TYPE_LIST,(anydata,BOOST_PP_NIL))'
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E2,BOOST_VMD_RETURN_TYPE_TUPLE) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(anydata,BOOST_PP_NIL))'
```

The SEQUENCE_MULTI_E2 is an invalid array, but valid tuple, followed by a valid list.

When we access element 0 of our sequence, and use a return type modifier that parses its data type as an array we will get UB. But if we use the return type modifiers BOOST_VMD_RETURN_TYPE_LIST or BOOST_VMD_RETURN_TYPE_TUPLE we are never parsing the invalid array as an array but as a tuple instead and therefore we successfully return the type of the invalid array as a BOOST_VMD_TYPE_TUPLE.

When we access element 1 of our sequence, which is both a valid list and a valid tuple, we will never get UB. We will get the return type of the element based on which return type modifier we use.

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

BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E3,BOOST_VMD_RETURN_TYPE) will return '(BOOST_VMD_TYPE_AR↵
RAY,(2,(3,4)))'
BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E3,BOOST_VMD_RETURN_TYPE_ARRAY) will return '(BOOST_VMD_TYPE_AR↵
RAY,(2,(3,4)))'
BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E3,BOOST_VMD_RETURN_TYPE_LIST) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(2,(3,4)))'
BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E3,BOOST_VMD_RETURN_TYPE_TUPLE) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(2,(3,4)))'
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E3,BOOST_VMD_RETURN_TYPE)
   will produce UB when attempting to parse the invalid list as a list
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E3,BOOST_VMD_RETURN_TYPE_ARRAY) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(anydata,BOOST_PP_NIL))'
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E3,BOOST_VMD_RETURN_TYPE_LIST)
   will produce UB when attempting to parse the invalid list as a list
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E3,BOOST_VMD_RETURN_TYPE_TUPLE) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(anydata,BOOST_PP_NIL))'
```

The SEQUENCE_MULTI_E3 is a valid array followed by a invalid list, but valid tuple.

When we access element 0 of our sequence, which is both a valid array and a valid tuple, we will never get UB. We will get the return type of the element based on which return type modifier we use.

When we access element 1 of our sequence, and use a return type modifier that parses its data type as a list we will get UB. But if we use the return type modifiers BOOST_VMD_RETURN_TYPE_ARRAY or BOOST_VMD_RETURN_TYPE_TUPLE we are never parsing the invalid list as a list but as a tuple instead and therefore we successfully return the type of the invalid list as a BOOST_VMD_TYPE_TUPLE.

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

BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E4,BOOST_VMD_RETURN_TYPE)
   will produce UB when attempting to parse the invalid array
BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E4,BOOST_VMD_RETURN_TYPE_ARRAY)
   will produce UB when attempting to parse the invalid array
BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E4,BOOST_VMD_RETURN_TYPE_LIST) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(2,(3,4)))'
BOOST_VMD_ELEM(0,SEQUENCE_MULTI_E4,BOOST_VMD_RETURN_TYPE_TUPLE) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(2,(3,4)))'
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E4,BOOST_VMD_RETURN_TYPE)
   will produce UB when attempting to parse the invalid list
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E4,BOOST_VMD_RETURN_TYPE_ARRAY) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(anydata,BOOST_PP_NIL))'
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E4,BOOST_VMD_RETURN_TYPE_LIST)
   will produce UB when attempting to parse the invalid list
BOOST_VMD_ELEM(1,SEQUENCE_MULTI_E4,BOOST_VMD_RETURN_TYPE_TUPLE) will re↵
turn '(BOOST_VMD_TYPE_TUPLE,(anydata,BOOST_PP_NIL))'
```

The SEQUENCE_MULTI_E4 is an invalid array, but valid tuple, followed by a invalid list, but valid tuple.

When we access element 0 of our sequence, which is an invalid array but a valid tuple, we must use a return type modifier which does not parse element 0 as an array, else we will get UB. This means we must use the return type modifiers BOOST_VMD_RE-

TURN_TYPE_LIST or BOOST_VMD_RETURN_TYPE_TUPLE so we are never parsing the invalid array as an array but as a tuple instead and therefore we successfully return the type of the invalid array as a BOOST_VMD_TYPE_TUPLE.

When we access element 1 of our sequence, which is an invalid list but a valid tuple, we must use a return type modifier which does not parse element 1 as a list, else we will get UB. This means we must use the return type modifiers BOOST_VMD_RETURN_TYPE_ARRAY or BOOST_VMD_RETURN_TYPE_TUPLE so we are never parsing the invalid list as a list but as a tuple instead and therefore we successfully return the type of the invalid list as a BOOST_VMD_TYPE_TUPLE.

### Usage with other modifiers of BOOST_VMD_ELEM

We have not yet discussed the rest of the modifiers which may be used with BOOST_VMD_ELEM, but return type modifiers are completely independent of any of them. This means they can be combined with other modifiers and whenever the element of the sequence is returned the return type modifiers determine of what the value of that element consists; whether it be just the element data or the element as a type/data tuple with the type parsed according to our return type modifier. When we subsequently discuss the use of other modifiers with BOOST_VMD_ELEM and refer to the element being returned, we are referring to that element as it is determined by the return type modifiers, which by default only returns the element's data.

## Filtering modifiers

Filtering modifiers are optional modifiers which work with some generic macros to specify a type of data to apply to the macro's functionality. The filtering modifier itself is an optional parameter specified as a v-type. Any v-type, specified as an optional parameter, may be used as a filtering modifier.

### Usage with equality macros

The equality macros, BOOST_VMD_EQUAL and BOOST_VMD_NOT_EQUAL, tests generically whether each of its two required inputs are equal or not equal to each other.

Each of these macro takes a single optional parameter, a filtering modifier, to narrow the focus of its equality testing to a particular v-type.

For the macro BOOST_VMD_EQUAL this optional parameter when specified means that equality is not only that the two required input parameters are equal but also that they are of the type or of a subtype of the third optional parameter. A number and a v-type are subtypes of identifiers while a non-empty list and an array are subtypes of tuples.

Conversely BOOST_VMD_NOT_EQUAL, with the optional third v-type parameter, returns 1 if either the first two parameters are not equal or if the type of the first two parameters is not the type or a subtype of the third parameter. Otherwise it returns 0. It is implemented as the complement of BOOST_VMD_EQUAL so that whenever BOOST_VMD_EQUAL returns 1, BOOST_VMD_NOT_EQUAL returns 0 and vice versa.

Here is an example of using BOOST_VMD_EQUAL with a filtering modifier. BOOST_VMD_NOT_EQUAL is just the complement of the results in our example for each result, and would be redundant to be specified each time below.

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

#define BOOST_VMD_REGISTER_AN_ID1 (AN_ID1)
#define BOOST_VMD_REGISTER_AN_ID2 (AN_ID2)

#define BOOST_VMD_DETECT_AN_ID1_AN_ID1
#define BOOST_VMD_DETECT_AN_ID2_AN_ID2

#define AN_IDENTIFIER1 AN_ID1
#define AN_IDENTIFIER2 AN_ID2
#define AN_IDENTIFIER3 AN_ID1 // same as AN_IDENTIFIER1 = AN_ID1

#define A_NUMBER1 33
#define A_NUMBER2 145
#define A_NUMBER3 33 // same as A_NUMBER1 = 33

#define A_TUPLE1 (AN_IDENTIFIER1,A_NUMBER1)
#define A_TUPLE2 (AN_IDENTIFIER1,A_NUMBER2)
#define A_TUPLE3 (AN_IDENTIFIER3,A_NUMBER3) // same as A_TUPLE1 = (AN_ID1,33)

#define A_LIST1 (A_NUMBER1,(A_NUMBER3,BOOST_PP_NIL))
#define A_LIST2 (A_NUMBER1,(A_NUMBER2,BOOST_PP_NIL))
#define A_LIST3 (A_NUMBER1,(A_NUMBER3,BOOST_PP_NIL))
#define A_LIST4 BOOST_PP_NIL // empty list
#define A_LIST5 BOOST_PP_NIL // empty list

BOOST_VMD_EQUAL(AN_IDENTIFIER1,AN_IDENTIFIER2,BOOST_VMD_TYPE_IDENTIFIER) will return 0
BOOST_VMD_EQUAL(AN_IDENTIFIER1,AN_IDENTIFIER3,BOOST_VMD_TYPE_IDENTIFIER) will return 1
BOOST_VMD_EQUAL(AN_IDENTIFIER1,AN_IDENTIFIER3,BOOST_VMD_TYPE_TYPE) will return 0 be↵
cause the type does not match

BOOST_VMD_EQUAL(A_NUMBER1,A_NUMBER2,BOOST_VMD_TYPE_NUMBER) will return 0
BOOST_VMD_EQUAL(A_NUMBER1,A_NUMBER3,BOOST_VMD_TYPE_NUMBER) will return 1
BOOST_VMD_EQUAL(A_NUMBER1,A_NUMBER3,BOOST_VMD_TYPE_IDENTIFIER) will return 1 because a num↵
ber is an identifier
BOOST_VMD_EQUAL(A_NUMBER1,A_NUMBER3,BOOST_VMD_TYPE_EMPTY) will return 0 be↵
cause the type does not match

BOOST_VMD_EQUAL(A_TUPLE1,A_TUPLE2,BOOST_VMD_TYPE_TUPLE) will return 0
BOOST_VMD_EQUAL(A_TUPLE1,A_TUPLE3,BOOST_VMD_TYPE_TUPLE) will return 1
BOOST_VMD_EQUAL(A_TUPLE1,A_TUPLE3,BOOST_VMD_TYPE_ARRAY) will return 0 be↵
cause the type does not match

BOOST_VMD_EQUAL(A_LIST1,A_LIST2,BOOST_VMD_TYPE_LIST) will return 0
BOOST_VMD_EQUAL(A_LIST1,A_LIST3,BOOST_VMD_TYPE_LIST) will return 1
BOOST_VMD_EQUAL(A_LIST1,A_LIST3,BOOST_VMD_TYPE_IDENTIFIER) will return 0 be↵
cause the type does not match
BOOST_VMD_EQUAL(A_LIST1,A_LIST3,BOOST_VMD_TYPE_TUPLE) will return 1 be↵
cause a non-empty list is also a tuple
BOOST_VMD_EQUAL(A_LIST4,A_LIST5,BOOST_VMD_TYPE_LIST) will return 1
BOOST_VMD_EQUAL(A_LIST4,A_LIST5,BOOST_VMD_TYPE_IDENTIFIER) will return 1 be↵
cause an empty list is also an identifier
BOOST_VMD_EQUAL(A_LIST4,A_LIST5,BOOST_VMD_TYPE_TUPLE) will return 0 be↵
cause an empty list is not a tuple
```

## Usage with BOOST_VMD_ELEM

As with the equality macros BOOST_VMD_ELEM allows one to perform filtering for the result. An optional parameter of a v-type can be used so that BOOST_VMD_ELEM returns its result only if the sequence element is of the v-type specified, else it fails to find the element in the same way that an element number which is outside the bounds of the sequence fails.

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

#define BOOST_VMD_REGISTER_ANAME (ANAME) // an identifier must always be registered to be found ↵
by VMD
#define A_SEQUENCE (1,2,3) 46 (list_data1,(list_data2,BOOST_PP_NIL)) BOOST_VMD_TYPE_SEQ ANAME

BOOST_VMD_ELEM(0,A_SEQUENCE) will return (1,2,3)
BOOST_VMD_ELEM(0,A_SEQUENCE,BOOST_VMD_TYPE_TUPLE) will return (1,2,3)
BOOST_VMD_ELEM(0,A_SEQUENCE,BOOST_VMD_TYPE_SEQ) will return emptiness

BOOST_VMD_ELEM(1,A_SEQUENCE) will return 46
BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_NUMBER) will return 46
BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER) will return 46 since a num↵
ber is also an identifier
BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_LIST) will return emptiness

BOOST_VMD_ELEM(2,A_SEQUENCE) will return (list_data1,(list_data2,BOOST_PP_NIL))
BOOST_VMD_ELEM(2,A_SEQUENCE,BOOST_VMD_TYPE_LIST) will return (list_data1,(list_data2,BOOST_PP_NIL))
BOOST_VMD_ELEM(2,A_SEQUENCE,BOOST_VMD_TYPE_TUPLE) will re↵
turn (list_data1,(list_data2,BOOST_PP_NIL)) since a list is also a tuple
BOOST_VMD_ELEM(2,A_SEQUENCE,BOOST_VMD_TYPE_TYPE) will return emptiness

BOOST_VMD_ELEM(3,A_SEQUENCE) will return BOOST_VMD_TYPE_SEQ
BOOST_VMD_ELEM(3,A_SEQUENCE,BOOST_VMD_TYPE_TYPE) will return BOOST_VMD_TYPE_SEQ
BOOST_VMD_ELEM(3,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER) will re↵
turn BOOST_VMD_TYPE_SEQ since a type is also an identifier
BOOST_VMD_ELEM(3,A_SEQUENCE,BOOST_VMD_TYPE_TUPLE) will return emptiness

BOOST_VMD_ELEM(4,A_SEQUENCE) will return ANAME
BOOST_VMD_ELEM(4,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER) will return ANAME
BOOST_VMD_ELEM(4,A_SEQUENCE,BOOST_VMD_TYPE_NUMBER) will return emptiness

BOOST_VMD_ELEM(0,BOOST_PP_NIL) will return BOOST_PP_NIL
BOOST_VMD_ELEM(0,BOOST_PP_NIL,BOOST_VMD_TYPE_LIST) will re↵
turn BOOST_PP_NIL since it is an empty list
BOOST_VMD_ELEM(0,BOOST_PP_NIL,BOOST_VMD_TYPE_IDENTIFIER) will return BOOST_PP_NIL since it a re↵
gistered identifier
BOOST_VMD_ELEM(0,BOOST_PP_NIL,BOOST_VMD_TYPE_TUPLE) will return emptiness
```

If you specify more than one v-type as a filtering modifier to BOOST_VMD_ELEM the last v-type becomes the filter.

Filtering with BOOST_VMD_ELEM denotes the type of the data expected when the particular element is found. Because filtering represents the type of the data requested, filtering modifiers and return type modifiers are mutually exclusive and any filtering modifier means that return type modifiers specified are ignored.

# Identifier modifiers

Identifier modifiers are optional parameters which specify a set of identifiers to search in order to look for a particular identifier match rather than just any identifier.

## Usage with BOOST_VMD_IS_IDENTIFIER

Once we have both registered and pre-detected an identifier we can test whether an identifier is a particular identifier using BOOST_VMD_IS_IDENTIFER and identifier modifiers. We do this by passing optional parameter(s) to BOOST_VMD_IS_IDENTIFER. The optional parameter(s) are either a single tuple of possible identifiers we are trying to match, or the individual identifiers themselves as separate parameters.

Using the optional parameter(s) with BOOST_VMD_IS_IDENTIFER we are asking not only if our input is any of the registered identifiers but also if it is one of a number of pre-detected identifiers.

As an example:

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

#define BOOST_VMD_REGISTER_yellow (yellow)
#define BOOST_VMD_REGISTER_green (green)
#define BOOST_VMD_REGISTER_blue (blue)
#define BOOST_VMD_REGISTER_red (red)

#define BOOST_VMD_DETECT_yellow_yellow
#define BOOST_VMD_DETECT_green_green
#define BOOST_VMD_DETECT_blue_blue

BOOST_VMD_IS_IDENTIFIER(some_input,yellow) // returns 1 only if 'some_input is 'yellow', else 0
BOOST_VMD_IS_IDENTIFIER(some_input,yellow,blue) // returns 1 only if 'some_input is 'yellow' or ↵
'blue', else 0
BOOST_VMD_IS_IDENTIFIER(some_input,(yellow,green)) // returns 1 if 'some_input' is 'yellow' or ↵
'green', else 0

BOOST_VMD_IS_IDENTIFIER(some_input,red)
   // always returns 0, even if 'some_input' is 'red' since 'red' has not been pre-detected
```

whereas

```
BOOST_VMD_IS_IDENTIFIER(some_input) // returns 1 if 'some_input' is 'red' since 'red' has been ↵
registered
```

If you invoke BOOST_VMD_IS_IDENTIFIER with the optional parameter(s), the invocation will only return 1 if the input matches one the identifier(s) of the optional parameters and the identifier it matches has been registered and pre-detected.

## Usage with BOOST_VMD_ELEM

When we use the specific filter modifier BOOST_VMD_TYPE_IDENTIFIER as an optional parameter of BOOST_VMD_ELEM we are asking for a particular element of a sequence as long as it is a VMD identifier. With that specific filter modifier BOOST_VMD_TYPE_IDENTIFIER we can use identifier modifiers to ask for a particular element of a sequence as long as it matches one of our identifier modifiers. If the specific filter modifier BOOST_VMD_TYPE_IDENTIFIER is not being used then all identifier modifiers are ignored.

The syntax for specifying identifier modifiers using BOOST_VMD_ELEM is exactly the same as the equivalent feature of the BOOST_VMD_IS_IDENTIFIER macro explained just previously. Optional parameters in the form of identifiers can be specified either singly any number of times or once as part of a tuple. For an identifier found as a sequence element to match against one of these possible identifiers, the possible identifiers must be both registered and pre-detected.

Since filter modifiers, which are v-types, are also identifiers, if you want to use v-types as identifier modifiers you must use the form which places all identifier modifiers as part of a tuple. Otherwise any v-types specified singly as optional parameters are seen as filter modifiers and never as identifier modifiers.

Let's see how this works:

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

#define BOOST_VMD_REGISTER_ANAME (ANAME)
#define BOOST_VMD_REGISTER_APLACE (APLACE)
#define BOOST_VMD_REGISTER_ACOUNTRY (ACOUNTRY)

#define BOOST_VMD_DETECT_ANAME_ANAME
#define BOOST_VMD_DETECT_APLACE_APLACE

#define A_SEQUENCE (1,2,3) ANAME 46 BOOST_VMD_TYPE_SEQ ACOUNTRY

BOOST_VMD_ELEM(1,A_SEQUENCE) will return 'ANAME'
BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER) will return 'ANAME'
BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER,APLACE,ACOUNTRY) will return emptiness
BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER,ANAME,APLACE,ACOUNTRY) will return 'ANAME'
BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER,(APLACE,ACOUNTRY,ANAME)) will return 'ANAME'

BOOST_VMD_ELEM(4,A_SEQUENCE) will return 'ACOUNTRY'
BOOST_VMD_ELEM(4,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER) will return 'ACOUNTRY'
BOOST_VMD_ELEM(4,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER,ACOUNTRY,ANAME)
   will return emptiness since ACOUNTRY is not pre-detected
```

Let us illustrate the case where VMD identifiers can be represented as either filter modifiers or identifier modifiers.

Using the sequence above:

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

BOOST_VMD_ELEM(3,A_SEQUENCE) will return the BOOST_VMD_TYPE_SEQ type
BOOST_VMD_ELEM(3,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER)
   will return the BOOST_VMD_TYPE_SEQ type since a type is an identifier
BOOST_VMD_ELEM(3,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFI↵
ER,BOOST_VMD_TYPE_SEQ,BOOST_VMD_TYPE_TUPLE) will return emptiness
```

The last use of our macro returns emptiness because if there is more than one type specified as an optional parameter the last type is chosen for filtering. Since the last type for type filtering is BOOST_VMD_TYPE_TUPLE and our fourth element is a v-type and not a tuple, emptiness is returned. The syntax does not specifying filtering with identifiers as might be supposed since BOOST_VMD_TYPE_SEQ and BOOST_VMD_TYPE_TUPLE are not treated as identifier modifiers but rather as additional filter modifiers.

In order to do filtering with an identifier and do it against various types themselves, since v-types are identifiers, we must use the tuple form to specify our identifier modifiers:

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

BOOST_VMD_ELEM(3,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER,(BOOST_VMD_TYPE_SEQ,BOOST_VMD_TYPE_TUPLE))
   will return BOOST_VMD_TYPE_SEQ
```

Now BOOST_VMD_TYPE_SEQ and BOOST_VMD_TYPE_TUPLE are treated as identifiers modifiers to match against.

# Splitting modifiers

The BOOST_VMD_ELEM macro, which by default just returns an element of a sequence, has a usage where you can have it return both the element and the remaining part of the sequence after the element, or even just the remaining part of the sequence after the element by itself. This offers a form of splitting the sequence on a particular element. When used to return the remaining part of a sequence the remaining data may subsequently be treated as a VMD sequence again.

To do this another set of optional modifiers are used which will be called 'splitting modifers'. These modifiers are:

- BOOST_VMD_RETURN_AFTER, which returns both the element information and the rest of the sequence after the element as a two-element tuple
- BOOST_VMD_RETURN_ONLY_AFTER, which returns only the rest of the sequence after the element specified
- BOOST_VMD_RETURN_NO_AFTER, this is the internal default which only returns the element itself. It need never be specified but may be used to override a previous splitting modifier specified as an optional parameter.

If more than one of the splitting modifiers are specified as optional parameters to BOOST_VMD_ELEM the last one specified is in effect.

The splitting modifers BOOST_VMD_RETURN_NO_AFTER and BOOST_VMD_RETURN_AFTER work with either return type modifiers or filtering modifiers if they are used. The splitting modifier BOOST_VMD_RETURN_ONLY_AFTER works with filtering modifiers if it is used and any return type modifiers will be ignored. Optional modifiers may occur in any order after the required parameters to BOOST_VMD_ELEM.

If BOOST_VMD_RETURN_AFTER is in effect and an element is not found, either because the element number is out of range for the sequence or because filtering does not match the element type, a tuple will still be returned but both its elements will be empty.

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

#define BOOST_VMD_REGISTER_ANAME (ANAME) // an identifier must always be registered to be found ↵
by VMD
#define A_SEQUENCE (1,2,3) 46 (list_data1,BOOST_PP_NIL) BOOST_VMD_TYPE_SEQ ANAME

BOOST_VMD_ELEM(2,A_SEQUENCE) will return '(list_data1,BOOST_PP_NIL)'
BOOST_VMD_ELEM(2,A_SEQUENCE,BOOST_VMD_RETURN_NO_AFTER) will return '(list_data1,BOOST_PP_NIL)'
BOOST_VMD_ELEM(2,A_SEQUENCE,BOOST_VMD_RETURN_AFTER) will re↵
turn '((list_data1,BOOST_PP_NIL),BOOST_VMD_TYPE_SEQ ANAME)'
BOOST_VMD_ELEM(2,A_SEQUENCE,BOOST_VMD_RETURN_ONLY_AFTER) will return 'BOOST_VMD_TYPE_SEQ ANAME'

BOOST_VMD_ELEM(5,A_SEQUENCE) will return emptiness
BOOST_VMD_ELEM(5,A_SEQUENCE,BOOST_VMD_RETURN_NO_AFTER) will return emptiness
BOOST_VMD_ELEM(5,A_SEQUENCE,BOOST_VMD_RETURN_AFTER) will return '(,)'
BOOST_VMD_ELEM(5,A_SEQUENCE,BOOST_VMD_RETURN_ONLY_AFTER) will return emptiness
```

Combining splitting modifiers with return type modifiers:

```
BOOST_VMD_ELEM(2,A_SEQUENCE,BOOST_VMD_RETURN_AFTER,BOOST_VMD_RETURN_TYPE) will re↵
turn '((BOOST_VMD_TYPE_LIST,(list_data1,BOOST_PP_NIL)),BOOST_VMD_TYPE_SEQ ANAME)'
```

Combining splitting modifiers with filtering modifiers:

```
BOOST_VMD_ELEM(2,A_SEQUENCE,BOOST_VMD_RETURN_AFTER,BOOST_VMD_TYPE_LIST) will re↵
turn '((list_data1,BOOST_PP_NIL),BOOST_VMD_TYPE_SEQ ANAME)'
```

# Index modifiers

Index modifiers can be used with the BOOST_VMD_ELEM macro when identifier modifiers are being used. Index modifiers take two values:

- BOOST_VMD_RETURN_INDEX, return an index as a number, starting with 0, of the particular identifier modifer which matched, as part of the output of the BOOST_VMD_ELEM macro. If no particular identifier modifier matches, return emptiness as part of the output. The index number is determined purely by the order in which identifier modifiers are specified as optional parameters to BOOST_VMD_ELEM, whether singly as individual optional parameters or as a tuple of identifier modifiers.
- BOOST_VMD_RETURN_NO_INDEX, do not return an index as part of the output. This is the default value and need only be used to override the BOOST_VMD_RETURN_INDEX value if it is specified.

The BOOST_VMD_RETURN_INDEX tells the programmer which one of the identifier modifiers matched the element's data as an index. Some macro programmers find this more useful for the purposes of macro branching logic than branching using the actual name of the identifier itself.

When the index modifier BOOST_VMD_RETURN_INDEX is specified, and identifier modifiers are specified along with the BOOST_VMD_TYPE_IDENTIFIER filter modifier, the output of BOOST_VMD_ELEM becomes a tuple of two elements. The first tuple element is the element matched and the last tuple element is the index, starting with 0, of the identifier modifier which matched. If an element is not matched both tuple elements are empty.

If the splitting modifier BOOST_VMD_RETURN_AFTER is also specified then the output is a tuple of three elements. The first tuple element is the element matched, the second tuple element is the rest of the sequence after the matching element, and the last tuple element is the numeric index. If an element is not matched then all three tuple elements are empty.

If identifier modifiers and the BOOST_VMD_TYPE_IDENTIFIER filter modifier are not specified as optional parameters, then if BOOST_VMD_RETURN_INDEX is specified it is ignored. If the splitting modifier BOOST_VMD_RETURN_ONLY_AFTER is specified, if BOOST_VMD_RETURN_INDEX is also specified it is ignored.

Let's see how this works:

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

#define BOOST_VMD_REGISTER_ANAME (ANAME)
#define BOOST_VMD_REGISTER_APLACE (APLACE)
#define BOOST_VMD_REGISTER_ACOUNTRY (ACOUNTRY)

#define BOOST_VMD_DETECT_ANAME_ANAME
#define BOOST_VMD_DETECT_APLACE_APLACE

#define A_SEQUENCE (1,2,3) ANAME (1)(2) 46

BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER) will return 'ANAME'
BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER,APLACE,ACOUNTRY) will return emptiness
BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER,BOOST_VMD_RETURN_INDEX,APLACE,ACOUN↵
TRY) will return (,)
BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER,BOOST_VMD_RETURN_INDEX,ANAME,APLACE,ACOUN↵
TRY) will return '(ANAME,0)'
BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER,BOOST_VMD_RETURN_INDEX,(APLACE,ACOUN↵
TRY,ANAME)) will return '(ANAME,2)'
```

Used with splitting modifiers:

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

BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER,BOOST_VMD_RETURN_INDEX,APLACE,ACOUN↵
TRY,BOOST_VMD_RETURN_AFTER) will return (,,)
BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER,BOOST_VMD_RETURN_INDEX,ANAME,APLACE,ACOUN↵
TRY,BOOST_VMD_RETURN_AFTER) will return '(ANAME,(1)(2) 46,0)'
BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER,BOOST_VMD_RETURN_INDEX,(APLACE,ACOUN↵
TRY,ANAME),BOOST_VMD_RETURN_AFTER) will return '(ANAME,(1)(2) 46,2)'

BOOST_VMD_ELEM(1,A_SEQUENCE,BOOST_VMD_TYPE_IDENTIFIER,BOOST_VMD_RETURN_INDEX,(APLACE,ACOUN↵
TRY,ANAME),BOOST_VMD_RETURN_ONLY_AFTER) will return '(1)(2) 46'
```

# Modifiers and the single-element sequence

A single element sequence is what we normally think of when working with macro data. It is a single type of macro data passed as an input parameter to some macro and processed as such.

In its basic form without modifiers BOOST_VMD_ELEM serves to just return a particular element of a sequence. For a single element sequence BOOST_VMD_ELEM with element number 0, just returns the single-element sequence itself. This does not offer much

functionality for our simple sequence. However with modifiers we can do things generically with our single-element sequence which correspond to working with a single type of data and extracting information about it.

With the return type modifier we can get the type of the data along with the data. Of course we can also use BOOST_VMD_GET_TYPE to retrieve just the type of data.

With our filter modifier we can retrieve the data only if it is a particular type, else retrieve emptiness.

With the identifier modifier we can retrieve an identifier only if it matches one or more other identifiers, else retrieve emptiness.

With our index modifier we can retrieve both our identifier and its numeric index if it matches one or more other identifiers, else retrieve a tuple of two empty elements if no match is found.

# Identifier subtypes

Identifiers are the low-level data types which macro programmers use to pass preprocessing data most often. As we have seen VMD has a system for registering and detecting identifiers so that they can be parsed as part of preprocessor data. This system also includes comparing identifiers for equality or inequality using BOOST_VMD_EQUAL/BOOST_VMD_NOT_EQUAL and matching identifiers using identifier modifiers in BOOST_VMD_IS_IDENTIFIER and BOOST_VMD_ELEM. Together these facilities provide a rich set of functionality for handling identifiers in macros.

Both numbers and v-types are subtypes of identifiers, and can both be individually recognized as data types of their own or worked with as identifiers using the identifier facilities already mentioned. Numbers, in particular, also have a rich set of functionality within the Boost PP library. As subtypes numbers and v-types can be used as filter modifiers and can be returned as specific types either when invoking BOOST_VMD_GET_TYPE or when using return type modifiers. Furthermore VMD recognizes their individual v-types, BOOST_VMD_TYPE_NUMBER and BOOST_VMD_TYPE_TYPE, as VMD data when parsing sequences.

It is possible for the end-user to define his own identifier subtype. This is called a "user-defined subtype". Once a user-define subtype is created all the generic type facilities of VMD which subtypes such as a number or a v-type possess is automatically available for that user-define subtype.

## Defining a subtype

In order to define a user-defined subtype a number of steps need to be followed. These steps will be explained in detail further below:

1. Register and pre-detect all identifiers of that subtype.

2. Register and pre-detect a v-type name for that subtype.

3. Subset register all identifiers of the subtype.

4. Subset register the v-type name for the subtype.

When we do the above, it is best to put all the macros in a single header file and always include that header file when we work generically with our user-defined subtype.

## Register and pre-detect all identifiers of that subtype

Registering and pre-detecting all of the identiifers of that subtype is exactly the same as registering and pre-detecting any identifier.

Let's create some identifiers based for use in the mythical "udef" library. We will put all our macros in the header file udef_vmd_macros.hpp.

We will need distinct names for the identifiers in our library, so we will append UDEF_ to our identifier names to make them unique. Our udef library deals in geometrical shapes so we will create a user-defined subtype which consists of identifiers for the various shapes our udef library can manipulate in their macros. So our identifier registrations and pre-detections placed in our header file will be:

```
#define BOOST_VMD_REGISTER_UDEF_CIRCLE (UDEF_CIRCLE)
#define BOOST_VMD_REGISTER_UDEF_SQUARE (UDEF_SQUARE)
#define BOOST_VMD_REGISTER_UDEF_TRIANGLE (UDEF_TRIANGLE)
#define BOOST_VMD_REGISTER_UDEF_HEXAGON (UDEF_HEXAGON)

#define BOOST_VMD_DETECT_UDEF_CIRCLE_UDEF_CIRCLE
#define BOOST_VMD_DETECT_UDEF_SQUARE_UDEF_SQUARE
#define BOOST_VMD_DETECT_UDEF_TRIANGLE_UDEF_TRIANGLE
#define BOOST_VMD_DETECT_UDEF_HEXAGON_UDEF_HEXAGON
```

## Register and pre-detect a v-type name for that subtype

We need to create a unique v-type name for our user-defined subtype. The name does not have to begin with BOOST_VMD_TYPE_ but it should be unique. Since BOOST_VMD_TYPE_ is the common beginning of all v-types we will use it for consistency but will append to it UDEF_SHAPES to give it a uniqueness which should not be duplicated:

```
#define BOOST_VMD_REGISTER_BOOST_VMD_TYPE_UDEF_SHAPES (BOOST_VMD_TYPE_UDEF_SHAPES)

#define BOOST_VMD_DETECT_BOOST_VMD_TYPE_UDEF_SHAPES_BOOST_VMD_TYPE_UDEF_SHAPES
```

## Subtype register all identifiers of the subtype

The macro to register an identifier subset starts with BOOST_VMD_SUBTYPE_REGISTER_ and you append to it each identifier in the subset. This is very much like the way you use the BOOST_VMD_REGISTER_ macro. The difference is that unlike the BOOST_VMD_REGISTER_ macro, which expands to a tuple whose single element is the identifier, the BOOST_VMD_SUBTYPE_REGISTER_ expands to a tuple of two elements where the first element is the subtype v-type and the second element is the identifier.

For our udef user-defined subtype this would be:

```
#define BOOST_VMD_SUBTYPE_REGISTER_UDEF_CIRCLE (BOOST_VMD_TYPE_UDEF_SHAPES,UDEF_CIRCLE)
#define BOOST_VMD_SUBTYPE_REGISTER_UDEF_SQUARE (BOOST_VMD_TYPE_UDEF_SHAPES,UDEF_SQUARE)
#define BOOST_VMD_SUBTYPE_REGISTER_UDEF_TRIANGLE (BOOST_VMD_TYPE_UDEF_SHAPES,UDEF_TRIANGLE)
#define BOOST_VMD_SUBTYPE_REGISTER_UDEF_HEXAGON (BOOST_VMD_TYPE_UDEF_SHAPES,UDEF_HEXAGON)
```

## Subtype register the v-type name for the subtype

Doing a subset register of the actual udef v-type is fairly easy once we understand how to register an identifier subset. The only particular thing to realize is that the type of any v-type is the v-type BOOST_VMD_TYPE_TYPE. So our subset register of our new v-type BOOST_VMD_TYPE_UDEF_SHAPES is:

```
#define BOOST_VMD_SUBTYPE_RE↵
GISTER_BOOST_VMD_TYPE_UDEF_SHAPES (BOOST_VMD_TYPE_TYPE,BOOST_VMD_TYPE_UDEF_SHAPES)
```

## Using our identifier subset

Once we have added all of the above object-like macros for defining our user-defined subtype to the udef_vmd_macros.hpp header file we have a new data type which we can use generically just like we can use numbers or v-types generically. It is important to include the header udef_vmd_macros.hpp in some translation unit whenever we need the VMD functionality for our new data type. So in our examples we will assume that an '#include udef_vmd_macros.hpp' precedes each example.

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

#define A_SEQUENCE UDEF_SQUARE
#define A_SEQUENCE2 217
#define A_SEQUENCE3 BOOST_VMD_TYPE_UDEF_SHAPES
#define A_SEQUENCE4 BOOST_VMD_TYPE_NUMBER

BOOST_VMD_GET_TYPE(A_SEQUENCE) will return 'BOOST_VMD_TYPE_UDEF_SHAPES'
BOOST_VMD_GET_TYPE(A_SEQUENCE2) will return 'BOOST_VMD_TYPE_NUMBER'
BOOST_VMD_GET_TYPE(A_SEQUENCE3) will return 'BOOST_VMD_TYPE_TYPE'
BOOST_VMD_GET_TYPE(A_SEQUENCE4) will return 'BOOST_VMD_TYPE_TYPE'
```

Here we see that when we use our BOOST_VMD_GET_TYPE macro on a single-element sequence which is one of our user-defined subtype values we correctly get back our user-defined subtype's v-type, just like we do when we ask for the type of a number. Also

when we use our BOOST_VMD_GET_TYPE macro on our user-defined subtype's v-type itself we correctly get back the type of all v-types, which is BOOST_VMD_TYPE_TYPE, just like we do when we ask for the type of the v-type of a number.

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

#define A_SEQUENCE5 (1,2) UDEF_TRIANGLE

BOOST_VMD_ELEM(1,A_SEQUENCE5,BOOST_VMD_RETURN_TYPE) will re↵
turn '(BOOST_VMD_TYPE_UDEF_SHAPES,UDEF_TRIANGLE)'
BOOST_VMD_ELEM(0,A_SEQUENCE5,BOOST_VMD_RETURN_TYPE) will return '(BOOST_VMD_TYPE_TUPLE,(1,2))'
```

Here we see that we can use the return type modifier to get back both the type and the value in a two-element tuple for our user-defined subtype just as we so for any other type.

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

#define A_SEQUENCE6 UDEF_CIRCLE
#define A_SEQUENCE7 168

BOOST_VMD_EQUAL(A_SEQUENCE6,UDEF_CIRCLE,BOOST_VMD_TYPE_UDEF_SHAPES) will return '1'
BOOST_VMD_EQUAL(A_SEQUENCE6,UDEF_CIRCLE,BOOST_VMD_TYPE_LIST) will return '0'
BOOST_VMD_EQUAL(A_SEQUENCE7,168,BOOST_VMD_TYPE_NUMBER) will return '1'
BOOST_VMD_EQUAL(A_SEQUENCE7,168,BOOST_VMD_TYPE_SEQ) will return '0'
```

Here we can see that we can use the filter modifier with our user-defined subtype's v-type just as we can do with any other v-type, such as the number v-type.

In all respects once we define our subtype and provide those definitions in a header file, our user-defined subtype acts like any other v-type in our system. Since VMD functionality is laregly based on being able to recognize the type of data in macro input being able to define another 'type', as an identifier subtype, which VMD understands has value for the macro programmer.

## Uniqueness of identifier subtype values and v-type

When we define a new identifier subtype we need to be careful that the values of that subtype and its actual v-type are unique identifiers within any translation unit. This is the main difference between just defining identifiers and defining an identifier subtype.

Recall that when we just register and pre-detect identifiers we will have no problems if the same identifiers already have been registered and pre-detected within the same translation unit. This is because we are just redefining the exact same macro if this is the case.

But with identifier subtypes, when we use the BOOST_VMD_SUBTYPE_REGISTER_ macro to associate our subtype's v-type with our subtype identifiers, we will have problems if someone else has also defined an identifier subtype using the same identifiers as we use since we will be redefining the same object-like macro name with a different expansion. Even if someone else has registered/pre-detected an identifier we are using for out subtype without defining a subtype based on that identifier we will be causing a problem defining our subtype because VMD macros which generically return the type of a sequence or sequence element will return our subtype as the type rather than just BOOST_VMD_TYPE_IDENTIFIER which some programmer might expect.

The gist of this is that if we define a user-defined subtype its identifiers need to be unique within a given translation unit, and yet unique names make it harder for an end-user to use macros more naturally. In our given example with the mythical udef library we used identifiers such as 'UDEF_CIRCLE' etc. instead of the more natural sounding CIRCLE. So with user-defined identifier subtypes we have a tradeoff; we need unique identifier names both for our subtype identifiers and the v-type for our subtype identifiers so as not to conflict with others who might be using identifier subtypes, but those unique names might make using macros less "natural" On the other hand, just registering/pre-detecting identifiers has no such problem. This is an issue of which any user, looking to create his own data type using VMD by defining user-defined subtypes, should be aware.

# Asserting and data types

The VMD macros for identifying data 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 data type, else a preprocessing 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 preprocessing 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 preprocessing 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 preprocessing 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 data types

The data 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
- identifier, BOOST_VMD_ASSERT_IS_IDENTIFIER
- 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
- type, BOOST_VMD_ASSERT_IS_TYPE

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/assert_is_empty.hpp> // BOOST_VMD_ASSERT_IS_EMPTY
#include <boost/vmd/assert_is_identifier.hpp> // BOOST_VMD_ASSERT_IS_IDENTIFIER
#include <boost/vmd/assert_is_number.hpp> // BOOST_VMD_ASSERT_IS_NUMBER
#include <boost/vmd/assert_is_array.hpp> // BOOST_VMD_ASSERT_IS_ARRAY
#include <boost/vmd/assert_is_list.hpp> // BOOST_VMD_ASSERT_IS_LIST
#include <boost/vmd/assert_is_seq.hpp> // BOOST_VMD_ASSERT_IS_SEQ
#include <boost/vmd/assert_is_tuple.hpp> // BOOST_VMD_ASSERT_IS_TUPLE
#include <boost/vmd/assert_is_type.hpp> // BOOST_VMD_ASSERT_IS_TYPE
```

## Assertions and VC++

The VC++ compiler has a quirk when dealing with BOOST_VMD_ASSERT and the data 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:

```
#include <boost/preprocessor/comparison/greater.hpp>
#include <boost/preprocessor/control/iif.hpp>
#include <boost/preprocessor/tuple/size.hpp>
#include <boost/vmd/assert_is_tuple.hpp>

#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

```
#include <boost/preprocessor/cat.hpp>
#include <boost/preprocessor/comparison/greater.hpp>
#include <boost/preprocessor/control/iif.hpp>
#include <boost/preprocessor/tuple/size.hpp>
#include <boost/vmd/assert_is_tuple.hpp>

#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.

# 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:

```
#include <boost/preprocessor/control/iif.hpp>
#include <boost/preprocessor/facilities/empty.hpp>

#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:

```
#include <boost/preprocessor/control/iif.hpp>
#include <boost/preprocessor/facilities/empty.hpp>

#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:

```
#include <boost/preprocessor/control/iif.hpp>
#include <boost/preprocessor/facilities/identity.hpp>

#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:

```
#include <boost/preprocessor/control/iif.hpp>
#include <boost/preprocessor/facilities/empty.hpp>

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

#define MACRO_CALL_IF_PARAMETER_1(parameter2) some_processing_using_both_parameters
```

or

```
#include <boost/preprocessor/control/iif.hpp>
#include <boost/preprocessor/facilities/identity.hpp>

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

#define MACRO_CALL_IF_PARAMETER_1(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 number of parameters. This is where using 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 when BOOST_VMD_EMPTY or BOOST_VMD_IDENTITY is the choice.

Now for our two examples we can have:

```
#include <boost/preprocessor/control/iif.hpp>
#include <boost/vmd/empty.hpp>

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

#define MACRO_CALL_IF_PARAMETER_1(parameter2) some_processing_using_parameters
```

or

```
#include <boost/preprocessor/control/iif.hpp>
#include <boost/vmd/identity.hpp>

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

#define MACRO_CALL_IF_PARAMETER_1(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:

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

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

Alternatively you can change MACRO_CHOICE so that its implementation and usage is:

```
#include <boost/preprocessor/control/iif.hpp>
#include <boost/vmd/identity.hpp>

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

#define CALLING_MACRO_CHOICE(parameter1,parameter2) \
    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>
```

# 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. If a compiler does not support variadic macro none of the macros in VMD are defined.

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.

# Boost PP re-entrant versions

Nearly all 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, take the next available BOOST_PP_WHILE iteration as their first parameter, and then 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:

Arrays

- BOOST_VMD_IS_ARRAY_D
- BOOST_VMD_IS_EMPTY_ARRAY_D
- BOOST_VMD_ASSERT_IS_ARRAY_D

Identifiers

- BOOST_VMD_IS_IDENTIFIER_D
- BOOST_VMD_ASSERT_IS_IDENTIFIER_D

Lists

- BOOST_VMD_IS_LIST_D
- BOOST_VMD_IS_EMPTY_LIST_D
- BOOST_VMD_ASSERT_IS_LIST_D

Sequences

- BOOST_VMD_ELEM_D
- BOOST_VMD_ENUM_D
- BOOST_VMD_EQUAL_D
- BOOST_VMD_GET_TYPE_D
- BOOST_VMD_IS_MULTI_D
- BOOST_VMD_IS_UNARY_D
- BOOST_VMD_NOT_EQUAL_D
- BOOST_VMD_SIZE_D
- BOOST_VMD_TO_ARRAY_D
- BOOST_VMD_TO_LIST_D
- BOOST_VMD_TO_SEQ_D
- BOOST_VMD_TO_TUPLE_D

Seqs

- BOOST_VMD_IS_SEQ_D

- BOOST_VMD_ASSERT_IS_SEQ_D

Types

- BOOST_VMD_IS_TYPE_D
- BOOST_VMD_ASSERT_IS_TYPE_D

Other

- BOOST_VMD_IS_PARENS_EMPTY_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 data types. By this I mean that a macro could be designed to handle different data 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 data types is very interesting.

Of course macros need to be usable by an end-user so the syntactical ability of sequences to represent different types of input data must be balanced against ease of use and understanding when using a macro. But because certain 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 data type. As an example if you design a macro where some input can either be a number, an identifier, or some other data 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 VMD data type. So designing a macro with data types in mind often means restricting data to parseable top-level 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 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 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.

# Version 1.7 to 1.8 conversion

Since the current version of VMD has been drastically changed to make it easier to use VMD functionality this section details equivalent functionality for previous version 1.7 VMD macros.

The changes in functionality involve the parsing of sequences. The equivalent to all the V1.7 functionality, which looks for various data types at the beginning of a sequence, is encompassed by the V1.8 macro BOOST_VMD_ELEM(0,sequence,...), where '0' is the first sequence element and 'sequence' is the sequence, with its optional parameters.

## Identifier

V1.7

BOOST_VMD_IDENTIFIER(sequence,keys,...) looked for an identifier at the beginning of a sequence and returned a 2-element tuple, where the first element is the matching index starting with 1, or 0 if no identifier is found, and the second tuple element is the rest of the sequence or emptiness if no identifier is found.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,(identifiers),BOOST_VMD_TYPE_IDENTIFIER,BOOST_VMD_RETURN_AFTER,BOOST_VMD_RETURN_INDEX) returns a 3-element tuple where the identifier found is the first tuple element, the rest of the sequence is the second tuple element, and the matching index, starting with 0, is the 3rd tuple element. If no identifier is found all elements of the returned tuple are empty.

V1.7

BOOST_VMD_BEGIN_IDENTIFIER(sequence,keys,...) looked for an identifier at the beginning of a sequence and returned the matching index starting with 1, or 0 if no identifier is found.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,(identifiers),BOOST_VMD_TYPE_IDENTIFIER,BOOST_VMD_RETURN_INDEX) returns a 2-element tuple where the identifier found is the first tuple element and the matching index, starting with 0, is the 2nd tuple element. If no identifier is found both elements of the returned tuple are empty.

V1.7

BOOST_VMD_AFTER_IDENTIFIER(sequence,keys,...) looked for an identifier at the beginning of a sequence and returned the rest of the sequence or emptiness if no identifier is found.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,(identifiers),BOOST_VMD_TYPE_IDENTIFIER,BOOST_VMD_RETURN_ONLY_AFTER) is the exact equivalent.

V1.7

BOOST_VMD_IS_BEGIN_IDENTIFIER(sequence,keys,...) returns 1 if input begins with an identifier, else 0 if it does not.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,(identifiers),BOOST_VMD_TYPE_IDENTIFIER) returns the identifier found, otherwise emptiness if not found. You can use BOOST_PP_COMPL(BOOST_VMD_IS_EMPTY(BOOST_VMD_ELEM(0,sequence,identifiers,BOOST_VMD_TYPE_IDENTIFIER))) as the exact equivalent.

## Number

V1.7

BOOST_VMD_NUMBER(sequence,...) looked for a number at the beginning of a sequence and returned a 2-element tuple, where the first element is the number and the second tuple element is the rest of the sequence. If no number is found both tuple elements are empty.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_NUMBER,BOOST_VMD_RETURN_AFTER) is the exact equivalent.

V1.7

BOOST_VMD_BEGIN_NUMBER(sequence,...) looked for a number at the beginning of a sequence and returned the number if found or emptiness if no number is found.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_NUMBER) is the exact equivalent.

V1.7

BOOST_VMD_AFTER_NUMBER(sequence,...) looked for a number at the beginning of a sequence and returned the rest of the sequence or emptiness if no number is found.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_NUMBER,BOOST_VMD_RETURN_ONLY_AFTER) is the exact equivalent.

V1.7

BOOST_VMD_IS_BEGIN_NUMBER(sequence,...) returns 1 if input begins with a number, else 0 if it does not.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_NUMBER) returns the number found, otherwise emptiness if not found. You can use BOOST_PP_COMPL(BOOST_VMD_IS_EMPTY(BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_NUMBER))) as the exact equivalent.

## Array

V1.7

BOOST_VMD_ARRAY(sequence) looked for an array at the beginning of a sequence and returned a 2-element tuple, where the first element is the array and the second tuple element is the rest of the sequence. If no array is found both tuple elements are empty.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_ARRAY,BOOST_VMD_RETURN_AFTER) is the exact equivalent.

V1.7

BOOST_VMD_BEGIN_ARRAY(sequence) looked for a array at the beginning of a sequence and returned the array if found or emptiness if no array is found.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_ARRAY) is the exact equivalent.

V1.7

BOOST_VMD_AFTER_ARRAY(sequence) looked for an array at the beginning of a sequence and returned the rest of the sequence or emptiness if no array is found.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_ARRAY,BOOST_VMD_RETURN_ONLY_AFTER) is the exact equivalent.

V1.7

BOOST_VMD_IS_BEGIN_ARRAY(sequence,...) returns 1 if input begins with an array, else 0 if it does not.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_ARRAY) returns the array found, otherwise emptiness if not found. You can use BOOST_PP_COMPL(BOOST_VMD_IS_EMPTY(BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_ARRAY))) as the exact equivalent.

## List

V1.7

BOOST_VMD_LIST(sequence) looked for an list at the beginning of a sequence and returned a 2-element tuple, where the first element is the list and the second tuple element is the rest of the sequence. If no list is found both tuple elements are empty.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_LIST,BOOST_VMD_RETURN_AFTER) is the exact equivalent.

V1.7

BOOST_VMD_BEGIN_LIST(sequence) looked for a list at the beginning of a sequence and returned the list if found or emptiness if no list is found.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_LIST) is the exact equivalent.

V1.7

BOOST_VMD_AFTER_LIST(sequence) looked for an list at the beginning of a sequence and returned the rest of the sequence or emptiness if no list is found.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_LIST,BOOST_VMD_RETURN_ONLY_AFTER) is the exact equivalent.

V1.7

BOOST_VMD_IS_BEGIN_LIST(sequence,...) returns 1 if input begins with an list, else 0 if it does not.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_LIST) returns the list found, otherwise emptiness if not found. You can use BOOST_PP_COMPL(BOOST_VMD_IS_EMPTY(BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_LIST))) as the exact equivalent.

## Seq

V1.7

BOOST_VMD_SEQ(sequence) looked for a seq at the beginning of a sequence and returned a 2-element tuple, where the first element is the seq and the second tuple element is the rest of the sequence. If no seq is found both tuple elements are empty.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_SEQ,BOOST_VMD_RETURN_AFTER) is the exact equivalent.

V1.7

BOOST_VMD_BEGIN_SEQ(sequence) looked for a seq at the beginning of a sequence and returned the seq if found or emptiness if no seq is found.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_SEQ) is the exact equivalent.

V1.7

BOOST_VMD_AFTER_SEQ(sequence) looked for an seq at the beginning of a sequence and returned the rest of the sequence or emptiness if no seq is found.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_SEQ,BOOST_VMD_RETURN_ONLY_AFTER) is the exact equivalent.

V1.7

BOOST_VMD_IS_BEGIN_SEQ(sequence,...) returns 1 if input begins with an seq, else 0 if it does not.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_SEQ) returns the seq found, otherwise emptiness if not found. You can use BOOST_PP_COMPL(BOOST_VMD_IS_EMPTY(BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_SEQ))) as the exact equivalent.

## Tuple

V1.7

BOOST_VMD_TUPLE(sequence) looked for an tuple at the beginning of a sequence and returned a 2-element tuple, where the first element is the tuple and the second tuple element is the rest of the sequence. If no tuple is found both tuple elements are empty.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_TUPLE,BOOST_VMD_RETURN_AFTER) is the exact equivalent.

V1.7

BOOST_VMD_BEGIN_TUPLE(sequence) looked for a tuple at the beginning of a sequence and returned the tuple if found or emptiness if no tuple is found.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_TUPLE) is the exact equivalent.

V1.7

BOOST_VMD_AFTER_TUPLE(sequence) looked for an tuple at the beginning of a sequence and returned the rest of the sequence or emptiness if no tuple is found.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_TUPLE,BOOST_VMD_RETURN_ONLY_AFTER) is the exact equivalent.

V1.7

BOOST_VMD_IS_BEGIN_TUPLE(sequence,...) returns 1 if input begins with an tuple, else 0 if it does not.

V1.8 equivalent

BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_TUPLE) returns the tuple found, otherwise emptiness if not found. You can use BOOST_PP_COMPL(BOOST_VMD_IS_EMPTY(BOOST_VMD_ELEM(0,sequence,BOOST_VMD_TYPE_TUPLE))) as the exact equivalent.

# Examples using VMD functionality

Examples of library use are always highly personal. Any given library employing macro programming can decide what macro facilities are needed based on the library itself and then decide if functionality in a macro library like VMD makes macro programming in that library easier. To that end the examples presented here are highly arbitrary and are just efforts to illustrate possible use of functionality of VMD features without worrying too much if those examples have any practical beneficial use in real programming situations. In these examples I have endeavored, therefore, to present macro programming "snippets" using VMD functionality rather than complete solutions to a given practical problem.

## Switch macro

In C++ there is a 'switch' statement which we can emulate in macro programming using VMD. For the macro emulation we will have as parameters to our macro:

1. A value, which can be any data type VMD can parse.

2. A tuple of calling values. These will be used when calling the matching macro.

3. Variadic parameters, each of which are tuples. Each tuple consists of two elements, the name of a value to match and the name of a macro to call. For the 'default' case the tuple is a single element which is the name of a macro to call. These are our equivalents to the C++ switch 'case' statements.

The macro looks like:

```
BOOST_VMD_SWITCH(value,calling_values,...)
```

We have to be careful not to parse the name of our macro to call in any way since this is a failing condition for BOOST_VMD_IS_EMPTY and subsequently for any parsing of input data we might want to do. Instead we will just extract the calling macro name and just call it, passing the calling values.

Our processing is:

1. Convert our variadic parameters to a tuple since access to tuple elements is easier.

2. Use a BOOST_PP_WHILE loop to find the matching value and extract the calling macro from it. We will use BOOST_VMD_EQUAL to find the matching value.

3. Call the calling macro with the calling values when we return from our BOOST_PP_WHILE loop.

Here is our code:

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

#if BOOST_PP_VARIADICS

#include <boost/preprocessor/cat.hpp>
#include <boost/preprocessor/arithmetic/inc.hpp>
#include <boost/preprocessor/comparison/equal.hpp>
#include <boost/preprocessor/control/expr_iif.hpp>
#include <boost/preprocessor/control/iif.hpp>
#include <boost/preprocessor/control/while.hpp>
#include <boost/preprocessor/tuple/elem.hpp>
#include <boost/preprocessor/tuple/enum.hpp>
#include <boost/preprocessor/facilities/expand.hpp>
#include <boost/preprocessor/tuple/replace.hpp>
#include <boost/preprocessor/tuple/size.hpp>
#include <boost/preprocessor/variadic/to_tuple.hpp>
#include <boost/preprocessor/variadic/size.hpp>
#include <boost/vmd/equal.hpp>
#include <boost/vmd/identity.hpp>
#include <boost/vmd/is_empty.hpp>

/*

  State index into state values

*/

#define BOOST_VMD_SWITCH_STATE_ELEM_INDEX 2
#define BOOST_VMD_SWITCH_STATE_ELEM_DEFAULT 4
#define BOOST_VMD_SWITCH_STATE_ELEM_RESULT 5

/*

  Retrieve the state value, never changes

*/

#define BOOST_VMD_SWITCH_STATE_GET_VALUE(state) \
 BOOST_PP_TUPLE_ELEM(0,state) \
/**/

/*

  Retrieve the state tuple of values, never changes

*/

#define BOOST_VMD_SWITCH_STATE_GET_CHOICES(state) \
 BOOST_PP_TUPLE_ELEM(1,state) \
/**/

/*

  Retrieve the state index

*/

#define BOOST_VMD_SWITCH_STATE_GET_INDEX(state) \
 BOOST_PP_TUPLE_ELEM(2,state) \
/**/

/*
```

```
  Retrieve the state tuple of values size, never changes

*/

#define BOOST_VMD_SWITCH_STATE_GET_SIZE(state) \
 BOOST_PP_TUPLE_ELEM(3,state) \
/**/

/*

  Retrieve the state default tuple

*/

#define BOOST_VMD_SWITCH_STATE_GET_DEFAULT(state) \
 BOOST_PP_TUPLE_ELEM(4,state) \
/**/

/*

  Retrieve the state result tuple

*/

#define BOOST_VMD_SWITCH_STATE_GET_RESULT(state) \
 BOOST_PP_TUPLE_ELEM(5,state) \
/**/

/*

  Retrieve the current value tuple

*/

#define BOOST_VMD_SWITCH_STATE_GET_CURRENT_CHOICE(state) \
 BOOST_PP_TUPLE_ELEM \
  ( \
  BOOST_VMD_SWITCH_STATE_GET_INDEX(state), \
  BOOST_VMD_SWITCH_STATE_GET_CHOICES(state) \
  ) \
/**/

/*

  Expands to the state

  value = value to compare against
  tuple = choices as a tuple of values
  size = size of tuple of values

  None of these ever change in the WHILE state

*/

#define BOOST_VMD_SWITCH_STATE_EXPAND(value,tuple,size) \
 (value,tuple,0,size,(0,),(,)) \
/**/

/*

  Expands to the WHILE state

  The state to our WHILE consists of a tuple of elements:
```

```
  1: value to compare against
  2: tuple of values. Each value is a value/macro pair or if the default just a macro
  3: index into the values
  4: tuple for default macro. 0 means no default macro, 1 means default macro and then second ↵
value is the default macro.
  5: tuple of result matched. Emptiness means no result yet specified, 0 means no match, 1 means ↵
match and second value is the matching macro.

*/

#define BOOST_VMD_SWITCH_STATE(value,...) \
 BOOST_VMD_SWITCH_STATE_EXPAND \
  ( \
  value, \
  BOOST_PP_VARIADIC_TO_TUPLE(__VA_ARGS__), \
  BOOST_PP_VARIADIC_SIZE(__VA_ARGS__) \
  ) \
/**/

/*

  Sets the state upon a successful match.

  macro = is the matching macro found

*/

#define BOOST_VMD_SWITCH_OP_SUCCESS(d,state,macro) \
 BOOST_PP_TUPLE_REPLACE_D \
  ( \
  d, \
  state, \
  BOOST_VMD_SWITCH_STATE_ELEM_RESULT, \
  (1,macro) \
  ) \
/**/

/*

  Sets the state upon final failure to find a match.

  def = default tuple macro, ignored

*/

#define BOOST_VMD_SWITCH_OP_FAILURE(d,state,def) \
 BOOST_PP_TUPLE_REPLACE_D \
  ( \
  d, \
  state, \
  BOOST_VMD_SWITCH_STATE_ELEM_RESULT, \
  (0,) \
  ) \
/**/

/*

  Increments the state index into the tuple values

*/

#define BOOST_VMD_SWITCH_OP_UPDATE_INDEX(d,state) \
```

```
 BOOST_PP_TUPLE_REPLACE_D \
  ( \
  d, \
  state, \
  BOOST_VMD_SWITCH_STATE_ELEM_INDEX, \
  BOOST_PP_INC(BOOST_VMD_SWITCH_STATE_GET_INDEX(state)) \
  ) \
/**/

/*

  Choose our current value's macro as our successful match

  tuple = current tuple to test

*/

#define BOOST_VMD_SWITCH_OP_TEST_CURRENT_VALUE_MATCH(d,state,tuple) \
 BOOST_VMD_SWITCH_OP_SUCCESS(d,state,BOOST_PP_TUPLE_ELEM(1,tuple)) \
/**/

/*

  Update our state index

  tuple = current tuple to test, ignored

*/

#define BOOST_VMD_SWITCH_OP_TEST_CURRENT_VALUE_UPDATE_INDEX(d,state,tuple) \
 BOOST_VMD_SWITCH_OP_UPDATE_INDEX(d,state) \
/**/

/*

  Test our current value against our value to compare against

  tuple = current tuple to test

*/

#define BOOST_VMD_SWITCH_OP_TEST_CURRENT_VALUE(d,state,tuple) \
 BOOST_PP_IIF \
  ( \
  BOOST_VMD_EQUAL_D \
   ( \
   d, \
   BOOST_VMD_SWITCH_STATE_GET_VALUE(state), \
   BOOST_PP_TUPLE_ELEM(0,tuple) \
   ), \
  BOOST_VMD_SWITCH_OP_TEST_CURRENT_VALUE_MATCH, \
  BOOST_VMD_SWITCH_OP_TEST_CURRENT_VALUE_UPDATE_INDEX \
  ) \
 (d,state,tuple) \
/**/

/*

  Set our default macro and update the index in our WHILE state

  tuple = current tuple to test

*/
```

```
#if BOOST_VMD_MSVC

#define BOOST_VMD_SWITCH_OP_TEST_CURRENT_CREATE_DEFAULT_NN(number,name) \
 (number,name) \
/**/

#define BOOST_VMD_SWITCH_OP_TEST_CURRENT_CREATE_DEFAULT(d,state,tuple) \
 BOOST_VMD_SWITCH_OP_UPDATE_INDEX \
  ( \
  d, \
  BOOST_PP_TUPLE_REPLACE_D \
   ( \
   d, \
   state, \
   BOOST_VMD_SWITCH_STATE_ELEM_DEFAULT, \
   BOOST_VMD_SWITCH_OP_TEST_CURRENT_CREATE_DEFAULT_NN(1,BOOST_PP_TUPLE_ENUM(tuple)) \
   ) \
  ) \
/**/

#else

#define BOOST_VMD_SWITCH_OP_TEST_CURRENT_CREATE_DEFAULT(d,state,tuple) \
 BOOST_VMD_SWITCH_OP_UPDATE_INDEX \
  ( \
  d, \
  BOOST_PP_TUPLE_REPLACE_D \
   ( \
   d, \
   state, \
   BOOST_VMD_SWITCH_STATE_ELEM_DEFAULT, \
   (1,BOOST_PP_TUPLE_ENUM(tuple)) \
   ) \
  ) \
/**/

#endif

/*

  If our current value is a default macro, just set the default macro,
  else test our current value.

  tuple = current tuple to test

*/

#define BOOST_VMD_SWITCH_OP_TEST_CURRENT_TUPLE(d,state,tuple) \
 BOOST_PP_IIF \
  ( \
  BOOST_PP_EQUAL_D \
   ( \
   d, \
   BOOST_PP_TUPLE_SIZE(tuple), \
   1 \
   ), \
  BOOST_VMD_SWITCH_OP_TEST_CURRENT_CREATE_DEFAULT, \
  BOOST_VMD_SWITCH_OP_TEST_CURRENT_VALUE \
  ) \
 (d,state,tuple) \
/**/
```

```
/*

  Test the current value in our tuple of values

*/

#define BOOST_VMD_SWITCH_OP_TEST_CURRENT(d,state) \
 BOOST_VMD_SWITCH_OP_TEST_CURRENT_TUPLE \
  ( \
  d, \
  state, \
  BOOST_VMD_SWITCH_STATE_GET_CURRENT_CHOICE(state) \
  ) \
/**/

/*

  Choose the default macro as our successful match

  def = default tuple cosnisting of just the default macro name

*/

#define BOOST_VMD_SWITCH_OP_DEFAULT_RET_CHOSEN(d,state,def) \
 BOOST_VMD_SWITCH_OP_SUCCESS \
  ( \
  d, \
  state, \
  BOOST_PP_TUPLE_ELEM(1,def) \
  ) \
/**/

/*

  If the default macro exists, choose it else indicate no macro was found

  def = default tuple cosnisting of just the default macro name

*/

#define BOOST_VMD_SWITCH_OP_DEFAULT_RET(d,state,def) \
 BOOST_PP_IIF \
  ( \
  BOOST_PP_TUPLE_ELEM(0,def), \
  BOOST_VMD_SWITCH_OP_DEFAULT_RET_CHOSEN, \
  BOOST_VMD_SWITCH_OP_FAILURE \
  ) \
 (d,state,def) \
/**/

/*

 Try to choose the default macro if it exists

*/

#define BOOST_VMD_SWITCH_OP_DEFAULT(d,state) \
 BOOST_VMD_SWITCH_OP_DEFAULT_RET \
  ( \
  d, \
  state, \
  BOOST_VMD_SWITCH_STATE_GET_DEFAULT(state) \
  ) \
```

```
/**/

/*

  WHILE loop operation

  Check for the next value match or try to choose the default if all matches have been checked

*/

#define BOOST_VMD_SWITCH_OP(d,state) \
 BOOST_PP_IIF \
  ( \
  BOOST_PP_EQUAL_D \
   (  \
   d, \
   BOOST_VMD_SWITCH_STATE_GET_INDEX(state), \
   BOOST_VMD_SWITCH_STATE_GET_SIZE(state) \
   ), \
  BOOST_VMD_SWITCH_OP_DEFAULT, \
  BOOST_VMD_SWITCH_OP_TEST_CURRENT \
  ) \
 (d,state) \
/**/

/*

  WHILE loop predicate

  Continue the WHILE loop if a result has not yet been specified

*/

#define BOOST_VMD_SWITCH_PRED(d,state) \
 BOOST_VMD_IS_EMPTY \
  ( \
  BOOST_PP_TUPLE_ELEM \
   ( \
   0, \
   BOOST_VMD_SWITCH_STATE_GET_RESULT(state) \
   ) \
  ) \
/**/

/*

  Invokes the function-like macro

  macro = function-like macro name
  tparams = tuple of macro parameters

*/

#define BOOST_VMD_SWITCH_PROCESS_INVOKE_MACRO(macro,tparams) \
 BOOST_PP_EXPAND(macro tparams) \
/**/

/*

  Processes our WHILE loop result

  callp = tuple of parameters for the called macro
  result = tuple. The first tuple element is 0
```

```
          if no macro has been found or 1 if a macro
          has been found. If 1 the second element is
          the name of a function-like macro

*/

#define BOOST_VMD_SWITCH_PROCESS(callp,result) \
 BOOST_PP_EXPR_IIF \
  ( \
  BOOST_PP_TUPLE_ELEM(0,result), \
  BOOST_VMD_SWITCH_PROCESS_INVOKE_MACRO \
   ( \
   BOOST_PP_TUPLE_ELEM(1,result), \
   callp \
   ) \
  ) \
/**/

/*

  Use BOOST_VMD_SWITCH_IDENTITY to pass a fixed value instead
  of a function-like macro as the second element of
  any tuple of the variadic parameters, or as the default
  value, to BOOST_VMD_SWITCH.

*/

#if BOOST_VMD_MSVC
#define BOOST_VMD_SWITCH_IDENTITY(item) BOOST_PP_CAT(BOOST_VMD_IDENTITY(item),)
#else
#define BOOST_VMD_SWITCH_IDENTITY BOOST_VMD_IDENTITY
#endif

/*

  Switch macro

  Parameters are:

  value = value to compare against. May be any VMD data value.
  callp = tuple of parameters for the called macro
  variadic parameters = each parameter must be a tuple.
    Each tuple consists of a two-element tuple. The first element is
    a value, which may be any VMD data value, and the second element
    is the name of a function-like macro to be called if the value
    is equal to the value to compare against. For a default value
    the tuple is a single-element tuple which contains the name of
    a function-like macro to be called if no other value matches.

*/

#define BOOST_VMD_SWITCH(value,callp,...) \
 BOOST_VMD_SWITCH_PROCESS \
  ( \
  callp, \
  BOOST_VMD_SWITCH_STATE_GET_RESULT \
   ( \
   BOOST_PP_WHILE \
    ( \
    BOOST_VMD_SWITCH_PRED, \
    BOOST_VMD_SWITCH_OP, \
    BOOST_VMD_SWITCH_STATE(value,__VA_ARGS__) \
```

```
    ) \
   ) \
  ) \
/**/

#endif /* BOOST_PP_VARIADICS */
```

The code is fairly involved but it is commented so that it can be understood. There are a few workarounds for a VC++ preprocessor problem, which I discovered, having to do with passing the name of a function-like macro in a tuple.

The BOOST_VMD_SWITCH macro can be used with either macros to call or with fixed values to return. When specifying macros to call the macro name is the second element of the corresponding value-macro tuple, or in the 'default' case it is just the macro name itself. When specifying fixed values to return the macro 'name' is BOOST_VMD_SWITCH_IDENTITY(fixed_value), whether as the second element of the corresponding value-macro tuple or as the macro 'name' of the 'default' case. In the variadic parameters the user can mix macro names and fixed values as he likes.

Some simple examples:

```
#define BOOST_VMD_SWITCH_TEST_1(number) \
 test1_ ## number
/**/

#define BOOST_VMD_SWITCH_TEST_2(number) \
 test2_ ## number
/**/

#define BOOST_VMD_SWITCH_TEST_3(number) \
 test3_ ## number
/**/

#define BOOST_VMD_SWITCH_TEST_DEFAULT(number) \
 test_default_ ## number
/**/
```

We will use these simple macros in our calls to BOOST_VMD_SWITCH.

```
BOOST_VMD_SWITCH(1,
    (7),
    (BOOST_VMD_SWITCH_TEST_DEFAULT),
    (3,BOOST_VMD_SWITCH_TEST_3),
    (1,BOOST_VMD_SWITCH_TEST_1),
    (2,BOOST_VMD_SWITCH_TEST_2)
    )
```

Here our macro will return 'test1_7'.

Notice that 'cases' can be in any order.

```
BOOST_VMD_SWITCH(4,
    (7),
    (BOOST_VMD_SWITCH_TEST_DEFAULT),
    (2,BOOST_VMD_SWITCH_TEST_2),
    (1,BOOST_VMD_SWITCH_TEST_1),
    (3,BOOST_VMD_SWITCH_TEST_3)
    )
```

Here are macro uses the default case and returns 'test_default_7'.

```
BOOST_VMD_SWITCH(143,
    (7),
    (BOOST_VMD_SWITCH_TEST_DEFAULT),
    (1,BOOST_VMD_SWITCH_TEST_1),
    (2,BOOST_VMD_SWITCH_TEST_2),
    (3,BOOST_VMD_SWITCH_TEST_3),
    (143,BOOST_VMD_SWITCH_IDENTITY(55))
    )
```

This shows how the matching case can be a fixed_value as the macro 'name'.

```
BOOST_VMD_SWITCH(155,
    (7),
    (BOOST_VMD_SWITCH_IDENTITY(77)),
    (1,BOOST_VMD_SWITCH_TEST_1),
    (2,BOOST_VMD_SWITCH_TEST_2),
    (3,BOOST_VMD_SWITCH_TEST_3),
    (143,BOOST_VMD_SWITCH_IDENTITY(55))
    )
```

This shows how the default value can be a fixed_value as the macro 'name'.

```
BOOST_VMD_SWITCH(BOOST_VMD_TYPE_TUPLE,
    (7),
    (BOOST_VMD_SWITCH_TEST_DEFAULT),
    (BOOST_VMD_TYPE_TUPLE,BOOST_VMD_SWITCH_TEST_1),
    ((1,2,3),BOOST_VMD_SWITCH_TEST_3),
    (2,BOOST_VMD_SWITCH_TEST_2)
    )
```

This shows that the 'value' and each 'case' matching values can be different data types just as long as the types are one which VMD can parse.

There is more that can be done with the BOOST_VMD_SWITCH code but as it is I believe it could be useful for programmers writing macro code. For instance there is no checking that more than one 'case' value is the same. We could generate a BOOST_VMD_ASSERT if that were the situation. There is no concept of falling through to the next 'case' as their is when 'break' is not used at the bottom of a particular C++ 'case' statement. Nonetheless the example gives the macro programmer an idea of what can be done using the BOOST_VMD_EQUAL macro in treating data types generically, using BOOST_VMD_IS_EMPTY to test for emptiness and using BOOST_VMD_IDENTITY to generate a fixed value when a macro call is made.

## TTI inner template

As a more practical example, just to show the possible use of VMD functionality in current Boost code, I will briefly illustrate a change that could be made to the TTI library when using VMD functionality.

The Boost TTI library, of which the current developer of VMD is also the developer, specifies a way to introspect an inner class template of a class. The introspection can occur for an inner class template of specific template parameters.

In the library a macro is used to generate the metafunction which allows the introspection to work. The macro used is called BOOST_TTI_TEMPLATE. The macro has both a variadic version and a non-variadic version.

In the non-variadic version the macro always takes two parameters for introspecting for specific template parameters. The first parameter is the name of the template and the second parameter is an array of the specific template parameters ( with or without the parameter names themselves ). So for a class template of the form:

```
template <class X,int Y> class MyTemplate { ... code };
```

the non-variadic macro would be:

```
BOOST_TTI_TEMPLATE(MyTemplate,(2,(class,int))) // uses array
```

I chose a Boost PP array rather than a Boost PP seq or a Boost PP list as I felt the notation for specifying the template parameters was closer with the array than with the others. Choosing a Boost PP tuple was not an option since for non-variadic macros there is no way to automatically know the tuple size, so an array was preferred.

For the variadic version variadic parameters are used so the notation would be:

```
BOOST_TTI_TEMPLATE(MyTemplate,class,int) // uses variadic parameters
```

since this is the most natural notation.

But for compatibility with the non-variadic version the end-user with variadic macro support could also choose the Boost PP array form above.

Using VMD the variadic version could support any of the other Boost PP composite types for the specific template parameters, even though I feel that the variadic parameters form is easiest to use. In this scenario a user could specify:

```
BOOST_TTI_TEMPLATE(MyTemplate,(class,(int,BOOST_PP_NIL))) // use a list
```

or

```
BOOST_TTI_TEMPLATE(MyTemplate,(class)(int)) // use a seq
```

or

```
BOOST_TTI_TEMPLATE(MyTemplate,(class,int)) // use a tuple
```

The only change needed would be in the code which takes the second parameter and converts it to the final form used internally ( a Boost PP array ). This occurs in the macro BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS in the <boost/tti/detail/dtemplate_params.hpp> file. The code has two situations, one for VC++8 or below and one for all other compilers. For our example we will concentrate just on the one for all other compilers. You do not need to know what the code does internally to complete the creation of the appropriate metafunction to follow this example. The macro code in question looks like this:

```
#define BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS(trait,name,tpArray) \
  BOOST_TTI_DETAIL_HAS_MEMBER_WITH_TEMPLATE_SFINAE \
    ( \
      ( BOOST_PP_ADD(BOOST_PP_ARRAY_SIZE(tpArray),4), ( trait, name, 1, false, BOOST_PP_AR↵
RAY_ENUM(tpArray) ) )  \
    ) \
/**/
```

In this code we are taking the name of the metafunction ( trait ), the name of the template ( name ), and our specific template parameters ( tpArray ) and passing the information in the form of a Boost PP array to another macro, which will eventually create the metafunction which the end-user uses to test if such a class template exists within some enclosing class. Even if tpArray were a list, seq, or tuple we still want to pass the information internally to BOOST_TTI_DETAIL_HAS_MEMBER_WITH_TEMPLATE_SFINAE in the form you can see above, which is a Boost PP array. We don't need or want to change that internal representation.

The current code, used by both the non-variadic and variadic version of the BOOST_TTI_TEMPLATE template, assumes the 'tpArray' parameter is a Boost PP array. But if it could be a tuple, seq, or list in the variadic version the code could become, with the appropriate Boost PP and VMD header files:

```
#include <boost/preprocessor/arithmetic/add.hpp>
#include <boost/preprocessor/array/enum.hpp>
#include <boost/preprocessor/array/size.hpp>
#include <boost/preprocessor/control/expr_iif.hpp>
#include <boost/preprocessor/control/iif.hpp>
#include <boost/preprocessor/list/enum.hpp>
#include <boost/preprocessor/list/size.hpp>
#include <boost/preprocessor/seq/enum.hpp>
#include <boost/preprocessor/seq/size.hpp>
#include <boost/preprocessor/tuple/enum.hpp>
#include <boost/preprocessor/tuple/size.hpp>
#include <boost/vmd/identity.hpp>
#include <boost/vmd/is_array.hpp>
#include <boost/vmd/is_list.hpp>
#include <boost/vmd/is_seq.hpp>
#include <boost/vmd/is_tuple.hpp>

#if BOOST_PP_VARIADICS

#define BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS(trait,name,tpArray) \
  BOOST_TTI_DETAIL_HAS_MEMBER_WITH_TEMPLATE_SFINAE \
    ( \
      BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS_TYPE_CONCAT \
        ( \
        trait,name,tpArray, \
        BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS_TYPE(tpArray) \
        ) \
    ) \
/**/

#define BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS_TYPE(tpArray) \
  BOOST_VMD_IDENTITY_RESULT \
    ( \
      BOOST_PP_IIF \
        ( \
        BOOST_VMD_IS_ARRAY(tpArray), \
        BOOST_VMD_IDENTITY(ARRAY), \
        BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS_TYPE_LIST \
        ) \
      (tpArray) \
    ) \
/**/

#define BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS_TYPE_LIST(tpArray) \
  BOOST_VMD_IDENTITY_RESULT \
    ( \
      BOOST_PP_IIF \
        ( \
        BOOST_VMD_IS_LIST(tpArray), \
        BOOST_VMD_IDENTITY(LIST), \
        BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS_TYPE_SEQ \
        ) \
      (tpArray) \
    ) \
/**/

#define BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS_TYPE_SEQ(tpArray) \
  BOOST_VMD_IDENTITY_RESULT \
    ( \
      BOOST_PP_IIF \
        ( \
        BOOST_VMD_IS_SEQ(tpArray), \
        BOOST_VMD_IDENTITY(SEQ), \
```

```
        BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS_TYPE_TUPLE \
        ) \
      (tpArray) \
    ) \
/**/

#define BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS_TYPE_TUPLE(tpArray) \
  BOOST_VMD_IDENTITY_RESULT \
    ( \
      BOOST_PP_EXPR_IIF \
        ( \
        BOOST_VMD_IS_TUPLE(tpArray), \
        BOOST_VMD_IDENTITY(TUPLE) \
        ) \
    ) \
/**/

#define BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS_TYPE_CONCAT(trait,name,tpAr↵
ray,name) \
  ( BOOST_PP_ADD(BOOST_PP_ ## name ## _SIZE(tpAr↵
ray),4), ( trait, name, 1, false, BOOST_PP_ ## name ## _ENUM(tpArray) ) )  \
/**/

#else

#define BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS(trait,name,tpArray) \
  BOOST_TTI_DETAIL_HAS_MEMBER_WITH_TEMPLATE_SFINAE \
    ( \
      ( BOOST_PP_ADD(BOOST_PP_ARRAY_SIZE(tpArray),4), ( trait, name, 1, false, BOOST_PP_AR↵
RAY_ENUM(tpArray) ) )  \
    ) \
/**/

#endif
```

This of course gets more elaborate, but could be shortened considerably if we chose to use BOOST_VMD_GET_TYPE and the invented BOOST_VMD_SWITCH of our first example. We will assume in this second version of the code above that our BOOST_VMD_SWITCH macro has been #included from somewhere.

```
#include <boost/preprocessor/arithmetic/add.hpp>
#include <boost/preprocessor/array/enum.hpp>
#include <boost/preprocessor/array/size.hpp>
#include <boost/preprocessor/list/enum.hpp>
#include <boost/preprocessor/list/size.hpp>
#include <boost/preprocessor/seq/enum.hpp>
#include <boost/preprocessor/seq/size.hpp>
#include <boost/preprocessor/tuple/enum.hpp>
#include <boost/preprocessor/tuple/size.hpp>
#include <boost/vmd/get_type.hpp>

#if BOOST_PP_VARIADICS

#define BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS(trait,name,tpArray) \
  BOOST_TTI_DETAIL_HAS_MEMBER_WITH_TEMPLATE_SFINAE \
    ( \
      BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS_TYPE_CONCAT \
        ( \
        trait,name,tpArray, \
        BOOST_VMD_SWITCH \
          ( \
          BOOST_VMD_GET_TYPE(tpArray), \
          (1), \
          (BOOST_VMD_TYPE_ARRAY,BOOST_VMD_SWITCH_IDENTITY(ARRAY)), \
          (BOOST_VMD_TYPE_LIST,BOOST_VMD_SWITCH_IDENTITY(LIST)), \
          (BOOST_VMD_TYPE_SEQ,BOOST_VMD_SWITCH_IDENTITY(SEQ)), \
          (BOOST_VMD_TYPE_TUPLE,BOOST_VMD_SWITCH_IDENTITY(TUPLE)) \
          ) \
        ) \
    ) \
/**/

#define BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS_TYPE_CONCAT(trait,name,tpAr↵
ray,name) \
  ( BOOST_PP_ADD(BOOST_PP_ ## name ## _SIZE(tpAr↵
ray),4), ( trait, name, 1, false, BOOST_PP_ ## name ## _ENUM(tpArray) ) )  \
/**/

#else

#define BOOST_TTI_DETAIL_TRAIT_CALL_HAS_TEMPLATE_CHECK_PARAMS(trait,name,tpArray) \
  BOOST_TTI_DETAIL_HAS_MEMBER_WITH_TEMPLATE_SFINAE \
    ( \
      ( BOOST_PP_ADD(BOOST_PP_ARRAY_SIZE(tpArray),4), ( trait, name, 1, false, BOOST_PP_AR↵
RAY_ENUM(tpArray) ) )  \
    ) \
/**/

#endif
```

This is shorter and easier to understand. The '(1)' passed as the calling values to BOOST_VMD_SWITCH could just as well be '()' but VC8 has trouble with empty parentheses so I avoid it here.

In the case of the TTI, is such a change worth it to give more flexibility to the end-user ? In reality, because the variadic version of passing the specific template parameters as variadic data is syntactically easier to use than any of the Boost PP composite forms, I am actually happy enough with that use not to pursue the sort of functionality I presented in this example. But the example nonetheless shows the power of the VMD functionality for creating macros which add flexibility when the macro programmer feels he needs it for his library.

# Variadic Macro Data Reference

## 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];

The errstr defaults to BOOST_VMD_ASSERT_ERROR if not supplied. It is only relevant for VC++.

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/assert_is_array.hpp>

```
BOOST_VMD_ASSERT_IS_ARRAY(sequence)
BOOST_VMD_ASSERT_IS_ARRAY_D(d, sequence)
```

### Macro BOOST_VMD_ASSERT_IS_ARRAY

BOOST_VMD_ASSERT_IS_ARRAY — Asserts that the sequence is a Boost PP array.

# Synopsis

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

BOOST_VMD_ASSERT_IS_ARRAY(sequence)
```

### Description

The macro checks that the sequence is a Boost PP array. If it is not a Boost PP array, it forces a compiler error.

The macro normally checks for a Boost PP 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.

sequence = a possible Boost PP array.

returns = Normally the macro returns nothing. If the sequence is a Boost PP 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 sequence is not a Boost PP 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 sequence is not a Boost PP array.

## Macro BOOST_VMD_ASSERT_IS_ARRAY_D

BOOST_VMD_ASSERT_IS_ARRAY_D — Asserts that the sequence is a Boost PP array. Re-entrant version.

# Synopsis

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

BOOST_VMD_ASSERT_IS_ARRAY_D(d, sequence)
```

### Description

The macro checks that the sequence is a Boost PP array. If it is not a Boost PP array, it forces a compiler error.

The macro normally checks for a Boost PP 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.

d = The next available BOOST_PP_WHILE iteration. sequence = a possible Boost PP sequence.

returns = Normally the macro returns nothing. If the sequence is a Boost PP 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 sequence is not a Boost PP 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 sequence is not a Boost PP array.

# Header <boost/vmd/assert_is_empty.hpp>

```

BOOST_VMD_ASSERT_IS_EMPTY(...)
```

## Macro BOOST_VMD_ASSERT_IS_EMPTY

BOOST_VMD_ASSERT_IS_EMPTY — Asserts that the input is empty.

# Synopsis

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

BOOST_VMD_ASSERT_IS_EMPTY(...)
```

### Description

The macro checks to see if the input is empty or not. If it is not empty, it forces a compiler error.

The macro is a variadic macro taking any input. For the VC++8 compiler (VS2005) the macro takes a single parameter of input to check and not variadic data.

The macro normally checks for emptiness 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.

.... = variadic input, for VC++8 this must be a single parameter.

returns = Normally the macro returns nothing. If the input is empty, 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 input is not empty the macro forces a compiler error by outputting invalid C++. For all other compilers a compiler error is forced without producing output if the input is not empty.

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/assert_is_identifier.hpp>

```


BOOST_VMD_ASSERT_IS_IDENTIFIER(...)
BOOST_VMD_ASSERT_IS_IDENTIFIER_D(d, ...)
```

## Macro BOOST_VMD_ASSERT_IS_IDENTIFIER

BOOST_VMD_ASSERT_IS_IDENTIFIER — Asserts that the sequence is an identifier.

# Synopsis

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

BOOST_VMD_ASSERT_IS_IDENTIFIER(...)
```

### Description

The macro checks that the sequence is an identifier. If it is not an identifier, it forces a compiler error.

The macro normally checks for an identifier 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.

... = variadic parameters

The variadic parameters are:

sequence = A sequence to test as an identifier. ids (optional) = The data may take one of two forms: it is either one or more single identifiers or a single Boost PP tuple of identifiers.

returns = Normally the macro returns nothing. If the sequence is an identifier, nothing is output. If optional ids are specified, for the sequence to be an identifier it must be an identifier that matches one of the optional ids. For VC++, because there is no sure way of forcing a compiler error from within a macro without producing output, if the sequence is not an identifier the macro forces a compiler error by outputting invalid C++. For all other compilers a compiler error is forced without producing output if the sequence is not an identifier.

Identifiers are registered in VMD with:

#define BOOST_VMD_REG_XXX (XXX) where XXX is a v-identifier.

The identifier must be registered to be found.

Identifiers are pre-detected in VMD with:

#define BOOST_VMD_DETECT_XXX_XXX where XXX is an identifier.

If you specify optional ids and have not specified the detection of an optional id, that id will never match an identifier.

## Macro BOOST_VMD_ASSERT_IS_IDENTIFIER_D

BOOST_VMD_ASSERT_IS_IDENTIFIER_D — Asserts that the sequence is an identifier. Re-entrant version.

# Synopsis

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

BOOST_VMD_ASSERT_IS_IDENTIFIER_D(d, ...)
```

### Description

The macro checks that the sequence is an identifier. If it is not an identifier, it forces a compiler error.

The macro normally checks for an identifier 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.

d = The next available BOOST_PP_WHILE iteration. ... = variadic parameters

The variadic parameters are:

sequence = A sequence to test as an identifier. ids (optional) = The data may take one of two forms: it is either one or more single identifiers or a single Boost PP tuple of identifiers.

returns = Normally the macro returns nothing. If the sequence is an identifier, nothing is output. If optional ids are specified, for the sequence to be an identifier it must be an identifier that matches one of the optional ids. For VC++, because there is no sure way of forcing a compiler error from within a macro without producing output, if the sequence is not an identifier the macro forces a compiler error by outputting invalid C++. For all other compilers a compiler error is forced without producing output if the sequence is not an identifier.

Identifiers are registered in VMD with:

#define BOOST_VMD_REG_XXX (XXX) where XXX is a v-identifier.

The identifier must be registered to be found.

Identifiers are pre-detected in VMD with:

#define BOOST_VMD_DETECT_XXX_XXX where XXX is an identifier.

If you specify optional ids and have not specified the detection of an optional id, that id will never match an identifier.

# Header <boost/vmd/assert_is_list.hpp>

```
BOOST_VMD_ASSERT_IS_LIST(sequence)
BOOST_VMD_ASSERT_IS_LIST_D(d, sequence)
```

## Macro BOOST_VMD_ASSERT_IS_LIST

BOOST_VMD_ASSERT_IS_LIST — Asserts that the sequence is a Boost PP list.

# Synopsis

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

BOOST_VMD_ASSERT_IS_LIST(sequence)
```

### Description

The macro checks that the sequence is a Boost PP list. If it is not a Boost PP list, it forces a compiler error.

The macro normally checks for a Boost PP 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.

sequence = a possible Boost PP list.

returns = Normally the macro returns nothing. If the sequence is a Boost PP 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 sequence is not a Boost PP 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 Boost PP list.

## Macro BOOST_VMD_ASSERT_IS_LIST_D

BOOST_VMD_ASSERT_IS_LIST_D — Asserts that the sequence is a Boost PP list. Re-entrant version.

# Synopsis

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

BOOST_VMD_ASSERT_IS_LIST_D(d, sequence)
```

### Description

The macro checks that the sequence is a Boost PP list. If it is not a Boost PP list, it forces a compiler error.

The macro normally checks for a Boost PP 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.

d = The next available BOOST_PP_WHILE iteration. sequence = a possible Boost PP list.

returns = Normally the macro returns nothing. If the sequence is a Boost PP 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 sequence is not a Boost PP 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 Boost PP list.

# Header <boost/vmd/assert_is_number.hpp>

```
BOOST_VMD_ASSERT_IS_NUMBER(sequence)
```

## Macro BOOST_VMD_ASSERT_IS_NUMBER

BOOST_VMD_ASSERT_IS_NUMBER — Asserts that the sequence is a number.

# Synopsis

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

BOOST_VMD_ASSERT_IS_NUMBER(sequence)
```

### Description

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

The macro normally checks for a number 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.

sequence = a possible number.

returns = Normally the macro returns nothing. If the sequence is a number, 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 sequence is not a number the macro forces a compiler error by outputting invalid C++. For all other compilers a compiler error is forced without producing output if the sequence is not a number.

# Header <boost/vmd/assert_is_seq.hpp>

```
BOOST_VMD_ASSERT_IS_SEQ(sequence)
BOOST_VMD_ASSERT_IS_SEQ_D(d, sequence)
```

## Macro BOOST_VMD_ASSERT_IS_SEQ

BOOST_VMD_ASSERT_IS_SEQ — Asserts that the sequence is a Boost PP seq.

# Synopsis

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

BOOST_VMD_ASSERT_IS_SEQ(sequence)
```

### Description

The macro checks that the sequence is a Boost PP seq. If it is not a Boost PP seq, it forces a compiler error.

The macro normally checks for a Boost PP 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.

sequence = a possible Boost PP seq.

returns = Normally the macro returns nothing. If the sequence is a Boost PP 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 sequence is not a Boost PP 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 sequence is not a Boost PP seq.

## Macro BOOST_VMD_ASSERT_IS_SEQ_D

BOOST_VMD_ASSERT_IS_SEQ_D — Asserts that the sequence is a Boost PP seq. Re-entrant version.

# Synopsis

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

BOOST_VMD_ASSERT_IS_SEQ_D(d, sequence)
```

### Description

The macro checks that the sequence is a Boost PP seq. If it is not a Boost PP seq, it forces a compiler error.

The macro normally checks for a Boost PP 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.

d = The next available BOOST_PP_WHILE iteration. sequence = a possible Boost PP seq.

returns = Normally the macro returns nothing. If the sequence is a Boost PP 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 sequence is not a Boost PP 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 sequence is not a Boost PP seq.

# Header <boost/vmd/assert_is_tuple.hpp>

```
BOOST_VMD_ASSERT_IS_TUPLE(sequence)
```

## Macro BOOST_VMD_ASSERT_IS_TUPLE

BOOST_VMD_ASSERT_IS_TUPLE — Asserts that the sequence is a Boost PP tuple.

# Synopsis

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

BOOST_VMD_ASSERT_IS_TUPLE(sequence)
```

### Description

The macro checks that the sequence is a Boost PP tuple. If it is not a Boost PP tuple, it forces a compiler error.

The macro normally checks for a Boost PP 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.

sequence = a possible Boost PP tuple.

returns = Normally the macro returns nothing. If the sequence is a Boost PP 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 sequence is not a Boost PP 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 sequence is not a Boost PP tuple.

# Header <boost/vmd/assert_is_type.hpp>

```
BOOST_VMD_ASSERT_IS_TYPE(sequence)
BOOST_VMD_ASSERT_IS_TYPE_D(d, sequence)
```

## Macro BOOST_VMD_ASSERT_IS_TYPE

BOOST_VMD_ASSERT_IS_TYPE — Asserts that the sequence is a VMD type.

# Synopsis

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

BOOST_VMD_ASSERT_IS_TYPE(sequence)
```

### Description

The macro checks that the sequence is a VMD type. If it is not a VMD type, it forces a compiler error.

The macro normally checks for a VMD type 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.

sequence = a possible VMD type.

returns = Normally the macro returns nothing. If the sequence is a VMD type, 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 sequence is not a VMD type the macro forces a compiler error by outputting invalid C++. For all other compilers a compiler error is forced without producing output if the sequence is not a VMD type.

## Macro BOOST_VMD_ASSERT_IS_TYPE_D

BOOST_VMD_ASSERT_IS_TYPE_D — Asserts that the sequence is a VMD type. Re-entrant version.

# Synopsis

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

BOOST_VMD_ASSERT_IS_TYPE_D(d, sequence)
```

### Description

The macro checks that the sequence is a VMD type. If it is not a VMD type, it forces a compiler error.

The macro normally checks for a VMD type 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.

d = The next available BOOST_PP_WHILE iteration. sequence = a possible VMD type.

returns = Normally the macro returns nothing. If the sequence is a VMD type, 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 sequence is not a VMD type the macro forces a compiler error by outputting invalid C++. For all other compilers a compiler error is forced without producing output if the sequence is not a VMD type.

# Header <boost/vmd/elem.hpp>

```
BOOST_VMD_ELEM(elem, ...)
BOOST_VMD_ELEM_D(d, elem, ...)
```

## Macro BOOST_VMD_ELEM

BOOST_VMD_ELEM — Accesses an element of a sequence.

# Synopsis

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

BOOST_VMD_ELEM(elem, ...)
```

### Description

elem = A sequence element number. From 0 to sequence size - 1. ... = Variadic parameters.

The first variadic parameter is required and is the sequence to access. Further variadic parameters are all optional.

With no further variadic parameters the macro returns the particular element in the sequence. If the element number is outside the bounds of the sequence macro access fails and the macro turns emptiness.

Optional parameters determine what it means that an element is successfully accessed as well as what data is returned by the macro.

Filters: specifying a VMD type tells the macro to return the element only if it is of the VMD type specified, else macro access fails. If more than one VMD type is specified as an optional parameter the last one specified is the filter.

Matching Identifiers: If the filter is specified as the identifier type, BOOST_VMD_TYPE_IDENTIFIER, optional parameters which are identifiers specify that the element accessed must match one of the identifiers else access fails. The identifiers may be specified multiple times as single optional parameters or once as a tuple of identifier parameters. If the identifiers are specified as single optional parameters they cannot be any of the specific BOOST_VMD_ optional parameters in order to be recognized as matching identifiers. Normally this should never be the case. The only situation where this could occur is if the VMD types, which are filters, are used as matching identifiers; in this case the matching identifiers need to be passed as a tuple of identifier parameters so they are not treated as filters.

Filters and matching identifiers change what it means that an element is successfully accessed. They do not change what data is returned by the macro. The remaining optional parameters do not change what it means that an element is successfully accessed but they do change what data is returned by the macro.

Splitting: Splitting allows the macro to return the rest of the sequence after the element accessed.

If BOOST_VMD_RETURN_AFTER is specified the return is a tuple with the element accessed as the first tuple parameter and the rest of the sequence as the second tuple parameter. If element access fails both tuple parameters are empty.

If BOOST_VMD_RETURN_ONLY_AFTER is specified the return is the rest of the sequence after the element accessed found. If the element access fails the return is emptiness.

If BOOST_VMD_RETURN_NO_AFTER, the default, is specified no splitting occurs.

If more than one of the splitting identifiers are specified the last one specified determines the splitting.

Return Type: The element accessed can be changed to return both the type of the element as well as the element data with optional return type parameters. When a type is returned, the element accessed which is returned becomes a two-element tuple where the type of the element accessed is the first tuple element and the element data itself is the second tuple element. If the macro fails to access the element the element access returned is emptiness and not a tuple.

If BOOST_VMD_RETURN_NO_TYPE, the default, is specified no type is returned as part of the element accessed.

If BOOST_VMD_RETURN_TYPE is specified the specific type of the element is returned in the tuple.

If BOOST_VMD_RETURN_TYPE_ARRAY is specified an array type is returned if the element is an array, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data.

If BOOST_VMD_RETURN_TYPE_LIST is specified a list type is returned if the element is a list, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data.

If BOOST_VMD_RETURN_TYPE_TUPLE is specified a tuple type is returned for all tuple-like data, else the actual type is returned for non-tuple data.

If more than one return type optional parameter is specified the last one specified determines the return type.

If a filter is specified optional return type parameters are ignored and the default BOOST_VMD_RETURN_NO_TYPE is in effect.

Index: If the filter is specified as the identifier type, BOOST_VMD_TYPE_IDENTIFIER, and matching identifiers are specified, an index parameter specifies that the numeric index, starting with 0, of the matching identifier found, be returned as part of the result.

If BOOST_VMD_RETURN_INDEX is specified an index is returned as part of the result.

If BOOST_VMD_RETURN_NO_INDEX, the default, is specified no index is returned as part of the result.

If both are specified the last one specified determines the index parameter.

When an index is returned as part of the result, the result is a tuple where the element accessed is the first tuple parameter and the index is the last tuple parameter. If element access fails the index is empty. If there is no BOOST_VMD_TYPE_IDENTIFIER filter or if there are no matching identifiers the BOOST_VMD_RETURN_INDEX is ignored and no index is returned as part of the result.

returns = With no optional parameters the element accessed is returned, or emptiness if element is outside the bounds of the sequence. Filters and matching identifiers can change the meaning of whether the element accessed is returned or failure occurs, but whenever failure occurs emptiness is returned as the element access part of that failure, else the element accessed is returned. Return type optional parameters, when filters are not used, return the element accessed as a two-element tuple where the first tuple element is the type and the second tuple element is the data; if the element is not accessed then emptiness is returned as the element access and not a tuple. Splitting with BOOST_VMD_RETURN_AFTER returns a tuple where the element accessed is the first tuple element and the rest of the sequence is the second tuple element. Splitting with BOOST_VMD_RETURN_ONLY_AFTER returns the rest of the sequence after the element accessed or emptiness if the element can not be accessed. Indexing returns the index as part of the output only if filtering with BOOST_VMD_TYPE_IDENTIFIER is specified and matching identifiers are specified. When the index is returned with BOOST_VMD_RETURN_AFTER it is the third element of the tuple returned, else it is the second element of a tuple where the element accessed is the first element of the tuple.

## Macro BOOST_VMD_ELEM_D

BOOST_VMD_ELEM_D — Accesses an element of a sequence. Re-entrant version.

# Synopsis

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

BOOST_VMD_ELEM_D(d, elem, ...)
```

## Description

d = The next available BOOST_PP_WHILE iteration. elem = A sequence element number. From 0 to sequence size - 1. ... = Variadic parameters.

The first variadic parameter is required and is the sequence to access. Further variadic parameters are all optional.

With no further variadic parameters the macro returns the particular element in the sequence. If the element number is outside the bounds of the sequence macro access fails and the macro turns emptiness.

Optional parameters determine what it means that an element is successfully accessed as well as what data is returned by the macro.

Filters: specifying a VMD type tells the macro to return the element only if it is of the VMD type specified, else macro access fails. If more than one VMD type is specified as an optional parameter the last one specified is the filter.

Matching Identifiers: If the filter is specified as the identifier type, BOOST_VMD_TYPE_IDENTIFIER, optional parameters which are identifiers specify that the element accessed must match one of the identifiers else access fails. The identifiers may be specified multiple times as single optional parameters or once as a tuple of identifier parameters. If the identifiers are specified as single optional parameters they cannot be any of the specific BOOST_VMD_ optional parameters in order to be recognized as matching identifiers. Normally this should never be the case. The only situation where this could occur is if the VMD types, which are filters, are used as matching identifiers; in this case the matching identifiers need to be passed as a tuple of identifier parameters so they are not treated as filters.

Filters and matching identifiers change what it means that an element is successfully accessed. They do not change what data is returned by the macro. The remaining optional parameters do not change what it means that an element is successfully accessed but they do change what data is returned by the macro.

Splitting: Splitting allows the macro to return the rest of the sequence after the element accessed.

If BOOST_VMD_RETURN_AFTER is specified the return is a tuple with the element accessed as the first tuple parameter and the rest of the sequence as the second tuple parameter. If element access fails both tuple parameters are empty.

If BOOST_VMD_RETURN_ONLY_AFTER is specified the return is the rest of the sequence after the element accessed found. If the element access fails the return is emptiness.

If BOOST_VMD_RETURN_NO_AFTER, the default, is specified no splitting occurs.

If more than one of the splitting identifiers are specified the last one specified determines the splitting.

Return Type: The element accessed can be changed to return both the type of the element as well as the element data with optional return type parameters. When a type is returned, the element accessed which is returned becomes a two-element tuple where the type of the element accessed is the first tuple element and the element data itself is the second tuple element. If the macro fails to access the element the element access returned is emptiness and not a tuple.

If BOOST_VMD_RETURN_NO_TYPE, the default, is specified no type is returned as part of the element accessed.

If BOOST_VMD_RETURN_TYPE is specified the specific type of the element is returned in the tuple.

If BOOST_VMD_RETURN_TYPE_ARRAY is specified an array type is returned if the element is an array, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data.

If BOOST_VMD_RETURN_TYPE_LIST is specified a list type is returned if the element is a list, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data.

If BOOST_VMD_RETURN_TYPE_TUPLE is specified a tuple type is returned for all tuple-like data, else the actual type is returned for non-tuple data. If more than one return type optional parameter is specified the last one specified determines the return type.

If a filter is specified optional return type parameters are ignored and the default BOOST_VMD_RETURN_NO_TYPE is in effect.

Index: If the filter is specified as the identifier type, BOOST_VMD_TYPE_IDENTIFIER, and matching identifiers are specified, an index parameter specifies that the numeric index, starting with 0, of the matching identifier found, be returned as part of the result.

If BOOST_VMD_RETURN_INDEX is specified an index is returned as part of the result.

If BOOST_VMD_RETURN_NO_INDEX, the default, is specified no index is returned as part of the result.

If both are specified the last one specified determines the index parameter.

When an index is returned as part of the result, the result is a tuple where the element accessed is the first tuple parameter and the index is the last tuple parameter. If element access fails the index is empty. If there is no BOOST_VMD_TYPE_IDENTIFIER filter or if there are no matching identifiers the BOOST_VMD_RETURN_INDEX is ignored and no index is returned as part of the result.

returns = With no optional parameters the element accessed is returned, or emptiness if element is outside the bounds of the sequence. Filters and matching identifiers can change the meaning of whether the element accessed is returned or failure occurs, but whenever failure occurs emptiness is returned as the element access part of that failure, else the element accessed is returned. Return type optional parameters, when filters are not used, return the element accessed as a two-element tuple where the first tuple element is the type and the second tuple element is the data; if the element is not accessed then emptiness is returned as the element access and not a tuple. Splitting with BOOST_VMD_RETURN_AFTER returns a tuple where the element accessed is the first tuple element and the rest of the sequence is the second tuple element. Splitting with BOOST_VMD_RETURN_ONLY_AFTER returns the rest of the sequence after the element accessed or emptiness if the element can not be accessed. Indexing returns the index as part of the output only if filtering with BOOST_VMD_TYPE_IDENTIFIER is specified and matching identifiers are specified. When the index is returned with BOOST_VMD_RETURN_AFTER it is the third element of the tuple returned, else it is the second element of a tuple where the element accessed is the first element of the tuple.

# Header <boost/vmd/empty.hpp>

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

## Macro BOOST_VMD_EMPTY

BOOST_VMD_EMPTY — Outputs emptiness.

# Synopsis

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

BOOST_VMD_EMPTY(...)
```

### Description

... = any variadic parameters. The parameters are ignored.

This macro is used to output emptiness ( nothing ) no matter what is passed to it.

If you use this macro to return a result, as in 'result BOOST_VMD_EMPTY' subsequently invoked, you should surround the result with BOOST_VMD_IDENTITY_RESULT to smooth over a VC++ problem.

# Header <boost/vmd/enum.hpp>

```
BOOST_VMD_ENUM(...)
BOOST_VMD_ENUM_D(d, ...)
```

## Macro BOOST_VMD_ENUM

BOOST_VMD_ENUM — Converts a sequence to comma-separated elements which are the elements of the sequence.

# Synopsis

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

BOOST_VMD_ENUM(...)
```

### Description

... = Variadic parameters.

The first variadic parameter is required and is the sequence to convert.

Further optional variadic parameters can be return type parameters. Return type parameters allow each element in the sequence to be converted to a two-element tuple where the first tuple element is the type and the second tuple element is the element data.

The BOOST_VMD_RETURN_NO_TYPE, the default, does not return the type as part of each converted element but just the data. All of the rest return the type and data as the two-element tuple. If BOOST_VMD_RETURN_TYPE is specified the specific type of the element is returned in the tuple. If BOOST_VMD_RETURN_TYPE_ARRAY is specified an array type is returned if the element is an array, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_LIST is specified a list type is returned if the element is a list, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_TUPLE is specified a tuple type is returned for all tuple-like data, else the actual type is returned for non-tuple data. If more than one return type optional parameter is specified the last one specified determines the return type.

returns = Comma-separated data, otherwise known as variadic data. If the sequence is empty the variadic data is empty. If an optional return type other than BOOST_VMD_RETURN_NO_TYPE is specified the type and the data of each element is returned as part of the variadic data. Otherwise just the data of each element is returned, which is the default.

## Macro BOOST_VMD_ENUM_D

BOOST_VMD_ENUM_D — Converts a sequence to comma-separated elements which are the elements of the sequence. Re-entrant version.

# Synopsis

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

BOOST_VMD_ENUM_D(d, ...)
```

### Description

d = The next available BOOST_PP_WHILE iteration. ... = Variadic parameters.

The first variadic parameter is required and is the sequence to convert.

Further optional variadic parameters can be return type parameters. Return type parameters allow each element in the sequence to be converted to a two-element tuple where the first tuple element is the type and the second tuple element is the element data.

The BOOST_VMD_RETURN_NO_TYPE, the default, does not return the type as part of each converted element but just the data. All of the rest return the type and data as the two-element tuple. If BOOST_VMD_RETURN_TYPE is specified the specific type of the element is returned in the tuple. If BOOST_VMD_RETURN_TYPE_ARRAY is specified an array type is returned if the element is an array, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_LIST is specified a list type is returned if the element is a list, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_TUPLE is specified a tuple type is returned for all tuple-like data, else the actual type is returned for non-tuple data. If more than one return type optional parameter is specified the last one specified determines the return type.

returns = Comma-separated data, otherwise known as variadic data. If the sequence is empty the variadic data is empty. If an optional return type other than BOOST_VMD_RETURN_NO_TYPE is specified the type and the data of each element is returned as part of the variadic data. Otherwise just the data of each element is returned, which is the default.

# Header <boost/vmd/equal.hpp>

```
BOOST_VMD_EQUAL(sequence, ...)
BOOST_VMD_EQUAL_D(d, sequence, ...)
```

## Macro BOOST_VMD_EQUAL

BOOST_VMD_EQUAL — Tests any two sequences for equality.

# Synopsis

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

BOOST_VMD_EQUAL(sequence, ...)
```

### Description

sequence = First sequence. ... = variadic parameters, maximum of 2.

The first variadic parameter is required and is the second sequence to test. The optional second variadic parameter is a VMD type as a filter.

The macro tests any two sequences for equality. For sequences to be equal the VMD types of each sequence must be equal and the individual elements of the sequence must be equal. For Boost PP composite types the macro tests that the composite types have the same size and then tests that each element of the composite type is equal. This means that all elements of a composite type must be a VMD type in order to use this macro successfully.

The single optional parameter is a filter. The filter is a VMD type which specifies that both sequences to test must be of that VMD type, as well as being equal to each other, for the test to succeed.

returns = 1 upon success or 0 upon failure. Success means that both sequences are equal and, if the optional parameter is specified, that the sequences are of the optional VMD type.

## Macro BOOST_VMD_EQUAL_D

BOOST_VMD_EQUAL_D — Tests any two sequences for equality. Re-entrant version.

# Synopsis

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

BOOST_VMD_EQUAL_D(d, sequence, ...)
```

### Description

d = The next available BOOST_PP_WHILE iteration. sequence = First sequence. ... = variadic parameters, maximum of 2.

The first variadic parameter is required and is the second sequence to test. The optional second variadic parameter is a VMD type as a filter.

The macro tests any two sequences for equality. For sequences to be equal the VMD types of each sequence must be equal and the individual elements of the sequence must be equal. For Boost PP composite types the macro tests that the composite types have the same size and then tests that each element of the composite type is equal. This means that all elements of a composite type must be a VMD type in order to use this macro successfully.

The single optional parameter is a filter. The filter is a VMD type which specifies that both sequences to test must be of that VMD type, as well as being equal to each other, for the test to succeed.

returns = 1 upon success or 0 upon failure. Success means that both sequences are equal and, if the optional parameter is specified, that the sequences are of the optional VMD type.

# Header <boost/vmd/get_type.hpp>

```


BOOST_VMD_GET_TYPE(...)
BOOST_VMD_GET_TYPE_D(d, ...)
```

## Macro BOOST_VMD_GET_TYPE

BOOST_VMD_GET_TYPE — Returns the type of a sequence as a VMD type.

# Synopsis

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

BOOST_VMD_GET_TYPE(...)
```

### Description

... = variadic parameters.

The first variadic parameter is required and is the sequence whose type we are getting.

The optional variadic parameters are return type parameters.

The macro returns the type of a sequence as a VMD type. The type of an empty sequence is always BOOST_VMD_TYPE_EMPTY and the type of a multi-element is always BOOST_VMD_TYPE_SEQUENCE. The type of a single-element sequence is the type of that single element.

The type returned can be modified by specifying an optional return type parameter.

If BOOST_VMD_RETURN_TYPE, the default, is specified the specific type of the element is returned.

If BOOST_VMD_RETURN_TYPE_ARRAY is specified an array type is returned if the element is an array, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data.

If BOOST_VMD_RETURN_TYPE_LIST is specified a list type is returned if the element is a list, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data.

If BOOST_VMD_RETURN_TYPE_TUPLE is specified a tuple type is returned for all tuple-like data, else the actual type is returned for non-tuple data.

If BOOST_VMD_RETURN_NO_TYPE is specified it is ignored since the macro always returns the type of the sequence.

If more than one return type optional parameter is specified the last one specified determines the return type.

returns = the type of the sequence as a VMD type.

## Macro BOOST_VMD_GET_TYPE_D

BOOST_VMD_GET_TYPE_D — Returns the type of a sequence as a VMD type. Re-entrant version.

# Synopsis

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

BOOST_VMD_GET_TYPE_D(d, ...)
```

### Description

d = The next available BOOST_PP_WHILE iteration. ... = variadic parameters.

The first variadic parameter is required and is the sequence whose type we are getting.

The optional variadic parameters are return type parameters.

The macro returns the type of a sequence as a VMD type. The type of an empty sequence is always BOOST_VMD_TYPE_EMPTY and the type of a multi-element is always BOOST_VMD_TYPE_SEQUENCE. The type of a single-element sequence is the type of that single element.

The type returned can be modified by specifying an optional return type parameter.

If BOOST_VMD_RETURN_TYPE, the default, is specified the specific type of the element is returned.

If BOOST_VMD_RETURN_TYPE_ARRAY is specified an array type is returned if the element is an array, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data.

If BOOST_VMD_RETURN_TYPE_LIST is specified a list type is returned if the element is a list, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data.

If BOOST_VMD_RETURN_TYPE_TUPLE is specified a tuple type is returned for all tuple-like data, else the actual type is returned for non-tuple data.

If BOOST_VMD_RETURN_NO_TYPE is specified it is ignored since the macro always returns the type of the sequence.

If more than one return type optional parameter is specified the last one specified determines the return type.

returns = the type of the sequence as a VMD type.

# Header <boost/vmd/identity.hpp>

```
BOOST_VMD_IDENTITY(item)
BOOST_VMD_IDENTITY_RESULT(result)
```

## Macro BOOST_VMD_IDENTITY

BOOST_VMD_IDENTITY — Macro which expands to its argument when invoked with any number of parameters.

# Synopsis

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

BOOST_VMD_IDENTITY(item)
```

### Description

item = any single argument

When BOOST_VMD_IDENTITY(item) is subsequently invoked with any number of parameters it expands to 'item'. Subsequently invoking the macro is done as 'BOOST_VMD_IDENTITY(item)(zero_or_more_arguments)'.

The macro is equivalent to the Boost PP macro BOOST_PP_IDENTITY(item) with the difference being that BOOST_PP_IDENTITY(item) is always invoked with no arguments, as in 'BOOST_VMD_IDENTITY(item)()' whereas BOOST_VMD_IDENTITY can be invoked with any number of arguments.

The macro is meant to be used in BOOST_PP_IF and BOOST_PP_IIF statements when only one of the clauses needs to be invoked with calling another macro and the other is meant to return an 'item'.

returns = the macro as 'BOOST_VMD_IDENTITY(item)', when invoked with any number of parameters as in '(zero_or_more_arguments)', returns 'item'. The macro itself returns 'item BOOST_VMD_EMPTY'.

## Macro BOOST_VMD_IDENTITY_RESULT

BOOST_VMD_IDENTITY_RESULT — Macro which wraps any result which can return its value using BOOST_VMD_IDENTITY or 'item BOOST_VMD_EMPTY'.

# Synopsis

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

BOOST_VMD_IDENTITY_RESULT(result)
```

### Description

result = any single result returned when BOOST_VMD_IDENTITY is used or 'item BOOST_VMD_EMPTY'.

The reason for this macro is to smooth over a problem when using VC++ with BOOST_VMD_IDENTITY. If your BOOST_VMD_IDENTITY macro can be used where VC++ is the compiler then you need to surround your macro code which could return a result with this macro in order that VC++ handles BOOST_VMD_IDENTITY correctly.

If you are not using VC++ you do not have to use this macro, but doing so does no harm.

# Header <boost/vmd/is_array.hpp>

```
BOOST_VMD_IS_ARRAY(sequence)
BOOST_VMD_IS_ARRAY_D(d, sequence)
```

## Macro BOOST_VMD_IS_ARRAY

BOOST_VMD_IS_ARRAY — Determines if a sequence is a Boost PP array.

# Synopsis

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

BOOST_VMD_IS_ARRAY(sequence)
```

### Description

The macro checks that the sequence is a Boost PP array. It returns 1 if it is an array, else if returns 0.

sequence = a possible Boost PP array.

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

The macro will generate a preprocessing error if the input is in the form of an array but its first tuple element, instead of being a number, is a preprocessor token which VMD cannot parse, as in the example '(&2,(0,1))' which is a valid tuple but an invalid array.

## Macro BOOST_VMD_IS_ARRAY_D

BOOST_VMD_IS_ARRAY_D — Determines if a sequence is a Boost PP array. Re-entrant version.

# Synopsis

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

BOOST_VMD_IS_ARRAY_D(d, sequence)
```

### Description

The macro checks that the sequence is a Boost PP array. It returns 1 if it is an array, else if returns 0.

d = The next available BOOST_PP_WHILE iteration. sequence = a possible Boost PP array.

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

The macro will generate a preprocessing error if the input is in the form of an array but its first tuple element, instead of being a number, is a preprocessor token which VMD cannot parse, as in the example '(&2,(0,1))' which is a valid tuple but an invalid array.

# Header <boost/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/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. For the VC++8 compiler (VS2005) the macro takes a single parameter of input to check.

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 PP library. 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, for VC++8 this must be a single parameter

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/is_empty_array.hpp>

```
BOOST_VMD_IS_EMPTY_ARRAY(sequence)
BOOST_VMD_IS_EMPTY_ARRAY_D(d, sequence)
```

## Macro BOOST_VMD_IS_EMPTY_ARRAY

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

# Synopsis

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

BOOST_VMD_IS_EMPTY_ARRAY(sequence)
```

### Description

An empty Boost PP array is a two element tuple where the first size element is 0 and the second element is a tuple with a single empty element, ie. '(0,())'.

sequence = a possible empty array

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

The macro will generate a preprocessing error if the sequence is in the form of an array but its first tuple element, instead of being a number, is a preprocessor token which VMD cannot parse, as in the example '(&0,())' which is a valid tuple but an invalid array.

## Macro BOOST_VMD_IS_EMPTY_ARRAY_D

BOOST_VMD_IS_EMPTY_ARRAY_D — Tests whether a sequence is an empty Boost PP array. Re-entrant version.

# Synopsis

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

BOOST_VMD_IS_EMPTY_ARRAY_D(d, sequence)
```

### Description

An empty Boost PP array is a two element tuple where the first size element is 0 and the second element is a tuple with a single empty element, ie. '(0,())'.

d = The next available BOOST_PP_WHILE iteration. sequence = a possible empty array

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

The macro will generate a preprocessing error if the sequence is in the form of an array but its first tuple element, instead of being a number, is a preprocessor token which VMD cannot parse, as in the example '(&0,())' which is a valid tuple but an invalid array.

# Header <boost/vmd/is_empty_list.hpp>

```


BOOST_VMD_IS_EMPTY_LIST(sequence)
BOOST_VMD_IS_EMPTY_LIST_D(d, sequence)
```

## Macro BOOST_VMD_IS_EMPTY_LIST

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

# Synopsis

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

BOOST_VMD_IS_EMPTY_LIST(sequence)
```

### 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.

sequence = a preprocessor parameter

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

The macro will generate a preprocessing error if the input as an empty list marker, instead of being an identifier, is a preprocessor token which VMD cannot parse, as in the example '&BOOST_PP_NIL'.

## Macro BOOST_VMD_IS_EMPTY_LIST_D

BOOST_VMD_IS_EMPTY_LIST_D — Tests whether a sequence is an empty Boost PP list. Re-entrant version.

# Synopsis

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

BOOST_VMD_IS_EMPTY_LIST_D(d, sequence)
```

### 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.

d = The next available BOOST_PP_WHILE iteration sequence = a preprocessor parameter

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

The macro will generate a preprocessing error if the input as an empty list marker, instead of being an identifier, is a preprocessor token which VMD cannot parse, as in the example '&BOOST_PP_NIL'.

# Header <boost/vmd/is_identifier.hpp>

```


BOOST_VMD_IS_IDENTIFIER(...)
BOOST_VMD_IS_IDENTIFIER_D(d, ...)
```

## Macro BOOST_VMD_IS_IDENTIFIER

BOOST_VMD_IS_IDENTIFIER — Tests whether a parameter is an identifier.

# Synopsis

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

BOOST_VMD_IS_IDENTIFIER(...)
```

### Description

... = variadic parameters

The first variadic parameter is required and it is the input to test.

Further variadic parameters are optional and are identifiers to match. The data may take one of two forms; it is either one or more single identifiers or a single Boost PP tuple of identifiers.

returns = 1 if the parameter is an identifier, otherwise 0. If the parameter is not an identifier, or if optional identifiers are specified and the identifier does not match any of the optional identifiers, the macro returns 0.

Identifiers are registered in VMD with:

#define BOOST_VMD_REG_XXX (XXX) where XXX is a v-identifier.

The identifier must be registered to be found.

Identifiers are pre-detected in VMD with:

#define BOOST_VMD_DETECT_XXX_XXX where XXX is an identifier.

If you specify optional identifiers and have not specified the detection of an optional identifier, that optional identifier will never match the input.

If the input is not a VMD data type this macro could lead to a preprocessor error. This is because the macro uses preprocessor concatenation to determine if the input is an identifier once it is determined that the input does not start with parenthesis. If the data being concatenated would lead to an invalid preprocessor token the compiler can issue a preprocessor error.

## Macro BOOST_VMD_IS_IDENTIFIER_D

BOOST_VMD_IS_IDENTIFIER_D — Tests whether a parameter is an identifier. Re-entrant version.

# Synopsis

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

BOOST_VMD_IS_IDENTIFIER_D(d, ...)
```

### Description

d = The next available BOOST_PP_WHILE iteration. ... = variadic parameters

The first variadic parameter is required and it is the input to test.

Further variadic parameters are optional and are identifiers to match. The data may take one of two forms; it is either one or more single identifiers or a single Boost PP tuple of identifiers.

returns = 1 if the parameter is an identifier, otherwise 0. If the parameter is not an identifier, or if optional identifiers are specified and the identifier does not match any of the optional identifiers, the macro returns 0.

Identifiers are registered in VMD with:

#define BOOST_VMD_REG_XXX (XXX) where XXX is a v-identifier.

The identifier must be registered to be found.

Identifiers are pre-detected in VMD with:

#define BOOST_VMD_DETECT_XXX_XXX where XXX is an identifier.

If you specify optional identifiers and have not specified the detection of an optional identifier, that optional identifier will never match the input.

If the input is not a VMD data type this macro could lead to a preprocessor error. This is because the macro uses preprocessor concatenation to determine if the input is an identifier once it is determined that the input does not start with parenthesis. If the data being concatenated would lead to an invalid preprocessor token the compiler can issue a preprocessor error.

# Header <boost/vmd/is_list.hpp>

```
BOOST_VMD_IS_LIST(sequence)
BOOST_VMD_IS_LIST_D(d, sequence)
```

## Macro BOOST_VMD_IS_LIST

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

# Synopsis

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

BOOST_VMD_IS_LIST(sequence)
```

### Description

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

sequence = input as a possible Boost PP list.

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

The macro will generate a preprocessing error if the input is in the form of a list but its end-of-list marker, instead of being an identifier, is a preprocessor token which VMD cannot parse, as in the example '(anything,&BOOST_PP_NIL)' which is a valid tuple but an invalid list.

## Macro BOOST_VMD_IS_LIST_D

BOOST_VMD_IS_LIST_D — Determines if a sequence is a Boost pplib list. Re-entrant version.

# Synopsis

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

BOOST_VMD_IS_LIST_D(d, sequence)
```

### Description

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

d = The next available BOOST_PP_WHILE iteration. sequence = input as a possible Boost PP list.

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

The macro will generate a preprocessing error if the input is in the form of a list but its end-of-list marker, instead of being an identifier, is a preprocessor token which VMD cannot parse, as in the example '(anything,&BOOST_PP_NIL)' which is a valid tuple but an invalid list.

# Header <boost/vmd/is_multi.hpp>

```
BOOST_VMD_IS_MULTI(sequence)
BOOST_VMD_IS_MULTI_D(d, sequence)
```

## Macro BOOST_VMD_IS_MULTI

BOOST_VMD_IS_MULTI — Determines if the sequence has more than one element, referred to as a multi-element sequence.

# Synopsis

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

BOOST_VMD_IS_MULTI(sequence)
```

### Description

sequence = a sequence

returns = 1 if the sequence is a multi-element sequence, else returns 0.

If the size of a sequence is known it is faster comparing that size to be greater than one to find out if the sequence is multi-element. But if the size of the sequence is not known it is faster calling this macro than getting the size and doing the previously mentioned comparison in order to determine if the sequence is multi-element or not.

## Macro BOOST_VMD_IS_MULTI_D

BOOST_VMD_IS_MULTI_D — Determines if the sequence has more than one element, referred to as a multi-element sequence.

# Synopsis

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

BOOST_VMD_IS_MULTI_D(d, sequence)
```

### Description

d = The next available BOOST_PP_WHILE iteration. sequence = a sequence

returns = 1 if the sequence is a multi-element sequence, else returns 0.

If the size of a sequence is known it is faster comparing that size to be greater than one to find out if the sequence is multi-element. But if the size of the sequence is not known it is faster calling this macro than getting the size and doing the previously mentioned comparison in order to determine if the sequence is multi-element or not.

# Header <boost/vmd/is_number.hpp>

```
BOOST_VMD_IS_NUMBER(sequence)
```

## Macro BOOST_VMD_IS_NUMBER

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

# Synopsis

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

BOOST_VMD_IS_NUMBER(sequence)
```

### Description

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

sequence = a possible number

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

If the input is not a VMD data type this macro could lead to a preprocessor error. This is because the macro uses preprocessor concatenation to determine if the input is a number once it is determined that the input does not start with parenthesis. If the data being concatenated would lead to an invalid preprocessor token the compiler can issue a preprocessor error.

# Header <boost/vmd/is_parens_empty.hpp>

```


BOOST_VMD_IS_PARENS_EMPTY(sequence)
BOOST_VMD_IS_PARENS_EMPTY_D(d, sequence)
```

## Macro BOOST_VMD_IS_PARENS_EMPTY

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

# Synopsis

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

BOOST_VMD_IS_PARENS_EMPTY(sequence)
```

### Description

sequence = a VMD sequence

returns = 1 if the sequence 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

## Macro BOOST_VMD_IS_PARENS_EMPTY_D

BOOST_VMD_IS_PARENS_EMPTY_D — Determines if the sequence is a set of parens with no data. Re-entrant version.

# Synopsis

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

BOOST_VMD_IS_PARENS_EMPTY_D(d, sequence)
```

### Description

d = The next available BOOST_PP_WHILE iteration. sequence = a VMD sequence

returns = 1 if the sequence 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/is_seq.hpp>

```


BOOST_VMD_IS_SEQ(sequence)
BOOST_VMD_IS_SEQ_D(d, sequence)
```

## Macro BOOST_VMD_IS_SEQ

BOOST_VMD_IS_SEQ — Determines if a sequence is a Boost PP seq.

# Synopsis

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

BOOST_VMD_IS_SEQ(sequence)
```

### Description

The macro checks that the sequence is a Boost PP seq. It returns 1 if it is a seq, else if returns 0.

sequence = a possible Boost PP seq

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

A single set of parentheses, with a single element, is parsed as a tuple and not a seq. To be parsed as a seq the input needs to be more than one consecutive sets of parentheses, each with a single element of data.

## Macro BOOST_VMD_IS_SEQ_D

BOOST_VMD_IS_SEQ_D — Determines if a sequence is a Boost PP seq. Re-entrant version.

# Synopsis

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

BOOST_VMD_IS_SEQ_D(d, sequence)
```

### Description

The macro checks that the sequence is a Boost PP seq. It returns 1 if it is a seq, else if returns 0.

d = The next available BOOST_PP_WHILE iteration. sequence = a possible Boost PP seq

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

A single set of parentheses, with a single element, is parsed as a tuple and not a seq. To be parsed as a seq the input needs to be more than one consecutive sets of parentheses, each with a single element of data.

# Header <boost/vmd/is_tuple.hpp>

```


BOOST_VMD_IS_TUPLE(sequence)
```

## Macro BOOST_VMD_IS_TUPLE

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

# Synopsis

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

BOOST_VMD_IS_TUPLE(sequence)
```

### Description

The macro checks to see if a sequence is a Boost PP tuple. A Boost PP tuple is preprocessor tokens enclosed by a set of parentheses with no preprocessing tokens before or after the parentheses.

sequence = a possible tuple

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

# Header <boost/vmd/is_type.hpp>

```


BOOST_VMD_IS_TYPE(sequence)
BOOST_VMD_IS_TYPE_D(d, sequence)
```

## Macro BOOST_VMD_IS_TYPE

BOOST_VMD_IS_TYPE — Tests whether a sequence is a VMD type.

# Synopsis

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

BOOST_VMD_IS_TYPE(sequence)
```

### Description

sequence = a possible VMD type

returns = 1 if the sequence is a VMD type, 0 if it is not.

If the sequence is not a VMD data type this macro could lead to a preprocessor error. This is because the macro uses preprocessor concatenation to determine if the sequence is an identifier once it is determined that the sequence does not start with parentheses. If the data being concatenated would lead to an invalid preprocessor token the compiler can issue a preprocessor error.

## Macro BOOST_VMD_IS_TYPE_D

BOOST_VMD_IS_TYPE_D — Tests whether a sequence is a VMD type. Re-entrant version.

# Synopsis

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

BOOST_VMD_IS_TYPE_D(d, sequence)
```

### Description

d = The next available BOOST_PP_WHILE iteration. sequence = a possible VMD type

returns = 1 if the sequence is a VMD type, 0 if it is not.

If the sequence is not a VMD data type this macro could lead to a preprocessor error. This is because the macro uses preprocessor concatenation to determine if the sequence is an identifier once it is determined that the sequence does not start with parentheses. If the data being concatenated would lead to an invalid preprocessor token the compiler can issue a preprocessor error.

# Header <boost/vmd/is_unary.hpp>

```


BOOST_VMD_IS_UNARY(sequence)
BOOST_VMD_IS_UNARY_D(d, sequence)
```

## Macro BOOST_VMD_IS_UNARY

BOOST_VMD_IS_UNARY — Determines if the sequence has only a single element, referred to as a single-element sequence.

# Synopsis

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

BOOST_VMD_IS_UNARY(sequence)
```

### Description

sequence = a VMD sequence

returns = 1 if the sequence is a single-element sequence, else returns 0.

If the size of a sequence is known it is faster comparing that size to be equal to one to find out if the sequence is single-element. But if the size of the sequence is not known it is faster calling this macro than getting the size and doing the previously mentioned comparison in order to determine if the sequence is single-element or not.

## Macro BOOST_VMD_IS_UNARY_D

BOOST_VMD_IS_UNARY_D — Determines if the sequence has only a single element, referred to as a single-element sequence. Re-entrant version.

# Synopsis

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

BOOST_VMD_IS_UNARY_D(d, sequence)
```

### Description

d = The next available BOOST_PP_WHILE iteration. sequence = a sequence

returns = 1 if the sequence is a single-element sequence, else returns 0.

If the size of a sequence is known it is faster comparing that size to be equal to one to find out if the sequence is single-element. But if the size of the sequence is not known it is faster calling this macro than getting the size and doing the previously mentioned comparison in order to determine if the sequence is single-element or not.

# Header <boost/vmd/not_equal.hpp>

```


BOOST_VMD_NOT_EQUAL(sequence, ...)
BOOST_VMD_NOT_EQUAL_D(d, sequence, ...)
```

## Macro BOOST_VMD_NOT_EQUAL

BOOST_VMD_NOT_EQUAL — Tests any two sequences for inequality.

# Synopsis

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

BOOST_VMD_NOT_EQUAL(sequence, ...)
```

### Description

sequence = First sequence. ... = variadic parameters, maximum of 2.

The first variadic parameter is required and is the second sequence to test. The optional second variadic parameter is a VMD type as a filter.

The macro tests any two sequences for inequality. For sequences to be unequal either the VMD types of each sequence must be unequal or the individual elements of the sequence must be unequal.

The single optional parameter is a filter. The filter is a VMD type which specifies that both sequences to test must be of that VMD type, as well as being equal to each other, for the test to fail, else it succeeds.

returns = 1 upon success or 0 upon failure. Success means that the sequences are unequal or, if the optional parameter is specified, that the sequences are not of the optional VMD type; otherwise 0 is returned if the sequences are equal.

The macro is implemented as the complement of BOOST_VMD_EQUAL, so that whenever BOOST_VMD_EQUAL would return 1 the macro returns 0 and whenever BOOST_VMD_EQUAL would return 0 the macro would return 1.

## Macro BOOST_VMD_NOT_EQUAL_D

BOOST_VMD_NOT_EQUAL_D — Tests any two sequences for inequality. Re-entrant version.

# Synopsis

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

BOOST_VMD_NOT_EQUAL_D(d, sequence, ...)
```

### Description

d = The next available BOOST_PP_WHILE iteration. sequence = First sequence. ... = variadic parameters, maximum of 2.

The first variadic parameter is required and is the second sequence to test. The optional second variadic parameter is a VMD type as a filter.

The macro tests any two sequences for inequality. For sequences to be unequal either the VMD types of each sequence must be unequal or the individual elements of the sequence must be unequal.

The single optional parameter is a filter. The filter is a VMD type which specifies that both sequences to test must be of that VMD type, as well as being equal to each other, for the test to fail, else it succeeds.

returns = 1 upon success or 0 upon failure. Success means that the sequences are unequal or, if the optional parameter is specified, that the sequences are not of the optional VMD type; otherwise 0 is returned if the sequences are equal.

The macro is implemented as the complement of BOOST_VMD_EQUAL, so that whenever BOOST_VMD_EQUAL would return 1 the macro returns 0 and whenever BOOST_VMD_EQUAL would return 0 the macro would return 1.

# Header <boost/vmd/size.hpp>

```
BOOST_VMD_SIZE(sequence)
BOOST_VMD_SIZE_D(d, sequence)
```

## Macro BOOST_VMD_SIZE

BOOST_VMD_SIZE — Returns the size of a sequence.

# Synopsis

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

BOOST_VMD_SIZE(sequence)
```

### Description

sequence = A sequence to test.

returns = If the sequence is empty returns 0, else returns the number of elements in the sequence.

## Macro BOOST_VMD_SIZE_D

BOOST_VMD_SIZE_D — Returns the size of a sequence. Re-entrant version.

# Synopsis

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

BOOST_VMD_SIZE_D(d, sequence)
```

### Description

d = The next available BOOST_PP_WHILE iteration. sequence = A sequence to test.

returns = If the sequence is empty returns 0, else returns the number of elements in the sequence.

# Header <boost/vmd/to_array.hpp>

```


BOOST_VMD_TO_ARRAY(...)
BOOST_VMD_TO_ARRAY_D(d, ...)
```

## Macro BOOST_VMD_TO_ARRAY

BOOST_VMD_TO_ARRAY — Converts a sequence to a Boost PP array whose elements are the elements of the sequence.

# Synopsis

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

BOOST_VMD_TO_ARRAY(...)
```

### Description

... = Variadic parameters.

The first variadic parameter is required and is the sequence to convert.

Further optional variadic parameters can be return type parameters. Return type parameters allow each element in the sequence to be converted to a two-element tuple where the first tuple element is the type and the second tuple element is the element data.

The BOOST_VMD_RETURN_NO_TYPE, the default, does not return the type as part of each converted element but just the data. All of the rest return the type and data as the two-element tuple. If BOOST_VMD_RETURN_TYPE is specified the specific type of the element is returned in the tuple. If BOOST_VMD_RETURN_TYPE_ARRAY is specified an array type is returned if the element is an array, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_LIST is specified a list type is returned if the element is a list, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_TUPLE is specified a tuple type is returned for all tuple-like data, else the actual type is returned for non-tuple data. If more than one return type optional parameter is specified the last one specified determines the return type.

returns = A Boost PP array. The sequence is empty the Boost PP array is an empty array. If an optional return type other than BOOST_VMD_RETURN_NO_TYPE is specified the type and the data of each element is returned as the array element. Otherwise just the data is returned as the array element, which is the default.

## Macro BOOST_VMD_TO_ARRAY_D

BOOST_VMD_TO_ARRAY_D — Converts a sequence to a Boost PP array whose elements are the elements of the sequence. Re-entrant version.

# Synopsis

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

BOOST_VMD_TO_ARRAY_D(d, ...)
```

### Description

d = The next available BOOST_PP_WHILE iteration. ... = Variadic parameters.

The first variadic parameter is required and is the sequence to convert.

Further optional variadic parameters can be return type parameters. Return type parameters allow each element in the sequence to be converted to a two-element tuple where the first tuple element is the type and the second tuple element is the element data.

The BOOST_VMD_RETURN_NO_TYPE, the default, does not return the type as part of each converted element but just the data. All of the rest return the type and data as the two-element tuple. If BOOST_VMD_RETURN_TYPE is specified the specific type of the element is returned in the tuple. If BOOST_VMD_RETURN_TYPE_ARRAY is specified an array type is returned if the element is an array, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_LIST is specified a list type is returned if the element is a list, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_TUPLE is specified a tuple type is returned for all tuple-like data, else the actual type is returned for non-tuple data. If more than one return type optional parameter is specified the last one specified determines the return type.

returns = A Boost PP array. The sequence is empty the Boost PP array is empty. If an optional return type other than BOOST_VMD_RETURN_NO_TYPE is specified the type and the data of each element is returned as the array element. Otherwise just the data is returned as the array element, which is the default.

# Header <boost/vmd/to_list.hpp>

```


BOOST_VMD_TO_LIST(...)
BOOST_VMD_TO_LIST_D(d, ...)
```

## Macro BOOST_VMD_TO_LIST

BOOST_VMD_TO_LIST — Converts a sequence to a Boost PP list whose elements are the elements of the sequence.

# Synopsis

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

BOOST_VMD_TO_LIST(...)
```

### Description

... = Variadic parameters.

The first variadic parameter is required and is the sequence to convert.

Further optional variadic parameters can be return type parameters. Return type parameters allow each element in the sequence to be converted to a two-element tuple where the first tuple element is the type and the second tuple element is the element data.

The BOOST_VMD_RETURN_NO_TYPE, the default, does not return the type as part of each converted element but just the data. All of the rest return the type and data as the two-element tuple. If BOOST_VMD_RETURN_TYPE is specified the specific type of the element is returned in the tuple. If BOOST_VMD_RETURN_TYPE_ARRAY is specified an array type is returned if the element is an array, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_LIST is specified a list type is returned if the element is a list, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_TUPLE is specified a tuple type is returned for all tuple-like data, else the actual type is returned for non-tuple data. If more than one return type optional parameter is specified the last one specified determines the return type.

returns = A Boost PP list. The sequence is empty the Boost PP list is an empty list. If an optional return type other than BOOST_VMD_RETURN_NO_TYPE is specified the type and the data of each element is returned as the list element. Otherwise just the data is returned as the list element, which is the default.

## Macro BOOST_VMD_TO_LIST_D

BOOST_VMD_TO_LIST_D — Converts a sequence to a Boost PP list whose elements are the elements of the sequence. Re-entrant version.

# Synopsis

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

BOOST_VMD_TO_LIST_D(d, ...)
```

### Description

d = The next available BOOST_PP_WHILE iteration. ... = Variadic parameters.

The first variadic parameter is required and is the sequence to convert.

Further optional variadic parameters can be return type parameters. Return type parameters allow each element in the sequence to be converted to a two-element tuple where the first tuple element is the type and the second tuple element is the element data.

The BOOST_VMD_RETURN_NO_TYPE, the default, does not return the type as part of each converted element but just the data. All of the rest return the type and data as the two-element tuple. If BOOST_VMD_RETURN_TYPE is specified the specific type of the element is returned in the tuple. If BOOST_VMD_RETURN_TYPE_ARRAY is specified an array type is returned if the element is an array, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If

BOOST_VMD_RETURN_TYPE_LIST is specified a list type is returned if the element is a list, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_TUPLE is specified a tuple type is returned for all tuple-like data, else the actual type is returned for non-tuple data. If more than one return type optional parameter is specified the last one specified determines the return type.

returns = A Boost PP list. The sequence is empty the Boost PP list is an empty list. If an optional return type other than BOOST_VMD_RETURN_NO_TYPE is specified the type and the data of each element is returned as the list element. Otherwise just the data is returned as the list element, which is the default.

# Header <boost/vmd/to_seq.hpp>

```
BOOST_VMD_TO_SEQ(...)
BOOST_VMD_TO_SEQ_D(d, ...)
```

## Macro BOOST_VMD_TO_SEQ

BOOST_VMD_TO_SEQ — Converts a sequence to a Boost PP seq whose elements are the elements of the sequence.

# Synopsis

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

BOOST_VMD_TO_SEQ(...)
```

### Description

... = Variadic parameters.

The first variadic parameter is required and is the sequence to convert.

Further optional variadic parameters can be return type parameters. Return type parameters allow each element in the sequence to be converted to a two-element tuple where the first tuple element is the type and the second tuple element is the element data.

The BOOST_VMD_RETURN_NO_TYPE, the default, does not return the type as part of each converted element but just the data. All of the rest return the type and data as the two-element tuple. If BOOST_VMD_RETURN_TYPE is specified the specific type of the element is returned in the tuple. If BOOST_VMD_RETURN_TYPE_ARRAY is specified an array type is returned if the element is an array, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_LIST is specified a list type is returned if the element is a list, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_TUPLE is specified a tuple type is returned for all tuple-like data, else the actual type is returned for non-tuple data. If more than one return type optional parameter is specified the last one specified determines the return type.

returns = A Boost PP seq. If the sequence is empty the return is emptiness since an empty seq does not exist. If an optional return type other than BOOST_VMD_RETURN_NO_TYPE is specified the type and the data of each element is returned as the seq element. Otherwise just the data is returned as the seq element, which is the default.

## Macro BOOST_VMD_TO_SEQ_D

BOOST_VMD_TO_SEQ_D — Converts a sequence to a Boost PP seq whose elements are the elements of the sequence. Re-entrant version.

# Synopsis

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

BOOST_VMD_TO_SEQ_D(d, ...)
```

## Description

d = The next available BOOST_PP_WHILE iteration. ... = Variadic parameters.

The first variadic parameter is required and is the sequence to convert.

Further optional variadic parameters can be return type parameters. Return type parameters allow each element in the sequence to be converted to a two-element tuple where the first tuple element is the type and the second tuple element is the element data.

The BOOST_VMD_RETURN_NO_TYPE, the default, does not return the type as part of each converted element but just the data. All of the rest return the type and data as the two-element tuple. If BOOST_VMD_RETURN_TYPE is specified the specific type of the element is returned in the tuple. If BOOST_VMD_RETURN_TYPE_ARRAY is specified an array type is returned if the element is an array, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_LIST is specified a list type is returned if the element is a list, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_TUPLE is specified a tuple type is returned for all tuple-like data, else the actual type is returned for non-tuple data. If more than one return type optional parameter is specified the last one specified determines the return type.

returns = A Boost PP seq. If the sequence is empty the return is emptiness since an empty seq does not exist. If an optional return type other than BOOST_VMD_RETURN_NO_TYPE is specified the type and the data of each element is returned as the seq element. Otherwise just the data is returned as the seq element, which is the default.

# Header <boost/vmd/to_tuple.hpp>

```


BOOST_VMD_TO_TUPLE(...)
BOOST_VMD_TO_TUPLE_D(d, ...)
```

## Macro BOOST_VMD_TO_TUPLE

BOOST_VMD_TO_TUPLE — Converts a sequence to a Boost PP tuple whose elements are the elements of the sequence.

# Synopsis

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

BOOST_VMD_TO_TUPLE(...)
```

## Description

... = Variadic parameters.

The first variadic parameter is required and is the sequence to convert.

Further optional variadic parameters can be return type parameters. Return type parameters allow each element in the sequence to be converted to a two-element tuple where the first tuple element is the type and the second tuple element is the element data.

The BOOST_VMD_RETURN_NO_TYPE, the default, does not return the type as part of each converted element but just the data. All of the rest return the type and data as the two-element tuple. If BOOST_VMD_RETURN_TYPE is specified the specific type of the element is returned in the tuple. If BOOST_VMD_RETURN_TYPE_ARRAY is specified an array type is returned if the element is an array, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_LIST is specified a list type is returned if the element is a list, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_TUPLE is specified a tuple type is returned for all tuple-like data, else the actual type is returned for non-tuple data. If more than one return type optional parameter is specified the last one specified determines the return type.

returns = A Boost PP tuple. If the sequence is empty the return is emptiness since an empty tuple does not exist. If an optional return type other than BOOST_VMD_RETURN_NO_TYPE is specified the type and the data of each element is returned as the tuple element. Otherwise just the data is returned as the tuple element, which is the default.

## Macro BOOST_VMD_TO_TUPLE_D

BOOST_VMD_TO_TUPLE_D — Converts a sequence to a Boost PP tuple whose elements are the elements of the sequence. Re-entrant version.

# Synopsis

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

BOOST_VMD_TO_TUPLE_D(d, ...)
```

### Description

d = The next available BOOST_PP_WHILE iteration. ... = Variadic parameters.

The first variadic parameter is required and is the sequence to convert.

Further optional variadic parameters can be return type parameters. Return type parameters allow each element in the sequence to be converted to a two-element tuple where the first tuple element is the type and the second tuple element is the element data.

The BOOST_VMD_RETURN_NO_TYPE, the default, does not return the type as part of each converted element but just the data. All of the rest return the type and data as the two-element tuple. If BOOST_VMD_RETURN_TYPE is specified the specific type of the element is returned in the tuple. If BOOST_VMD_RETURN_TYPE_ARRAY is specified an array type is returned if the element is an array, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_LIST is specified a list type is returned if the element is a list, else a tuple type is returned if the element is a tuple, else the actual type is returned for non-tuple data. If BOOST_VMD_RETURN_TYPE_TUPLE is specified a tuple type is returned for all tuple-like data, else the actual type is returned for non-tuple data. If more than one return type optional parameter is specified the last one specified determines the return type.

returns = A Boost PP tuple. If the sequence is empty the return is emptiness since an empty tuple does not exist. If an optional return type other than BOOST_VMD_RETURN_NO_TYPE is specified the type and the data of each element is returned as the tuple element. Otherwise just the data is returned as the tuple element, which is the default.

# 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 initial main design goal of this library was 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

On Windows I have tested this library using gcc/MingW, VC++, and clang targeting gcc. The compilers tested are gcc 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, 4.8.2, 4.8.3, 4.8.4, 4.9.0, 4.9.1, 4.9.2, VC++ 8.0, 9.0, 10.0, 11.0, 12.0, and the latest clang build from source.

On Linux I have tested this library using gcc 4.4.7, 4.6.4, 4.7.3, 4.8.2, 4.8.3, 4.9.1, clang 3.3, 3.4, 3.5, and the latest clang build from source, and Intel C++ 12.1. 13.0, 14.0, 15.0.

For VC++ 8.0 the BOOST_VMD_IS_EMPTY and BOOST_VMD_ASSERT_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 1.8

- After a review of VMD I have greatly simplified the main interfaces and added optional functionality in the form of modifiers. The main changes are the addition of the many generic macros for sequences and the expansions of types to include the v-type.
  - Added BOOST_VMD_ELEM macro.
  - Added BOOST_VMD_EQUAL macro.
  - Added BOOST_VMD_NOT_EQUAL macro.
  - Added BOOST_VMD_IS_MULTI macro.
  - Added BOOST_VMD_IS_TYPE macro.
  - Added BOOST_VMD_ASSERT_IS_TYPE macro.
  - Added BOOST_VMD_IS_UNARY macro.
  - Added BOOST_VMD_SIZE macro.
  - Replaced with the BOOST_VMD_ELEM macro, using modifiers, a number of macros which were eliminated. These are:
    - BOOST_VMD_IDENTIFER
    - BOOST_VMD_BEGIN_IDENTIFIER
    - BOOST_VMD_AFTER_IDENTIFIER
    - BOOST_VMD_IS_BEGIN_IDENTIFIER
    - BOOST_VMD_NUMBER
    - BOOST_VMD_BEGIN_NUMBER
    - BOOST_VMD_AFTER_NUMBER
    - BOOST_VMD_IS_BEGIN_NUMBER
    - BOOST_VMD_ARRAY
    - BOOST_VMD_BEGIN_ARRAY
    - BOOST_VMD_AFTER_ARRAY
    - BOOST_VMD_IS_BEGIN_ARRAY
    - BOOST_VMD_LIST
    - BOOST_VMD_BEGIN_LIST
    - BOOST_VMD_AFTER_LIST
    - BOOST_VMD_IS_BEGIN_LIST
    - BOOST_VMD_SEQ
    - BOOST_VMD_BEGIN_SEQ

  - BOOST_VMD_AFTER_SEQ
  - BOOST_VMD_IS_BEGIN_SEQ
  - BOOST_VMD_TUPLE
  - BOOST_VMD_BEGIN_TUPLE
  - BOOST_VMD_AFTER_TUPLE
  - BOOST_VMD_IS_BEGIN_TUPLE
- Every macro has its own header file.

## 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

BOOST_VMD_ASSERT_IS_NUMBER
Header < boost/vmd/assert_is_seq.hpp >
BOOST_VMD_ASSERT_IS_SEQ
BOOST_VMD_ASSERT_IS_SEQ_D
Header < boost/vmd/assert_is_tuple.hpp >
BOOST_VMD_ASSERT_IS_TUPLE
Header < boost/vmd/assert_is_type.hpp >
BOOST_VMD_ASSERT_IS_TYPE
BOOST_VMD_ASSERT_IS_TYPE_D
Header < boost/vmd/elem.hpp >
BOOST_VMD_ELEM
BOOST_VMD_ELEM_D
Header < boost/vmd/empty.hpp >
BOOST_VMD_EMPTY
Header < boost/vmd/enum.hpp >
BOOST_VMD_ENUM
BOOST_VMD_ENUM_D
Header < boost/vmd/equal.hpp >
BOOST_VMD_EQUAL
BOOST_VMD_EQUAL_D
Header < boost/vmd/get_type.hpp >
BOOST_VMD_GET_TYPE
BOOST_VMD_GET_TYPE_D
Header < boost/vmd/identity.hpp >
BOOST_VMD_IDENTITY
BOOST_VMD_IDENTITY_RESULT
Header < boost/vmd/is_array.hpp >
BOOST_VMD_IS_ARRAY
BOOST_VMD_IS_ARRAY_D
Header < boost/vmd/is_empty.hpp >
BOOST_VMD_IS_EMPTY
Header < boost/vmd/is_empty_array.hpp >
BOOST_VMD_IS_EMPTY_ARRAY
BOOST_VMD_IS_EMPTY_ARRAY_D
Header < boost/vmd/is_empty_list.hpp >
BOOST_VMD_IS_EMPTY_LIST
BOOST_VMD_IS_EMPTY_LIST_D
Header < boost/vmd/is_identifier.hpp >
BOOST_VMD_IS_IDENTIFIER
BOOST_VMD_IS_IDENTIFIER_D
Header < boost/vmd/is_list.hpp >
BOOST_VMD_IS_LIST
BOOST_VMD_IS_LIST_D
Header < boost/vmd/is_multi.hpp >
BOOST_VMD_IS_MULTI
BOOST_VMD_IS_MULTI_D
Header < boost/vmd/is_number.hpp >
BOOST_VMD_IS_NUMBER
Header < boost/vmd/is_parens_empty.hpp >
BOOST_VMD_IS_PARENS_EMPTY
BOOST_VMD_IS_PARENS_EMPTY_D
Header < boost/vmd/is_seq.hpp >
BOOST_VMD_IS_SEQ
BOOST_VMD_IS_SEQ_D
Header < boost/vmd/is_tuple.hpp >
BOOST_VMD_IS_TUPLE
Header < boost/vmd/is_type.hpp >
BOOST_VMD_IS_TYPE