Advanced Preprocessor Meta-Programming with Boost.Preprocessor Library

Asher Sterkin, NDS Technologies, Israel

Secure Enable Interact

I have no ambitions for improving C++ macros. Instead, I recommend the use of facilities from the C++ language proper, such as inline functions, templates, constructors (for initialization), destructors (for cleanup), exceptions (for exiting contexts), etc.

Bjarne Stroustrup’s C++ Style and Technique FAQ

Secure Enable Interact

Objectives

- Typical applications for preprocessor meta-programming
- Non-trivial programming techniques with Boost.Preprocessor

Agenda

- Why macros, anyhow?
- Variability points
- How can we test it?
- Typical applications:
  - Static Data Table Population
  - Bit Fields
  - IDL-like annotations

Why macros, anyhow?

- Not all compilers support advanced templates
- Synchronized definition of data and functions (DRY)
- Preserving naming convention (DRY)
- Source file location (__FILE__, __LINE__)

Don’t Repeat Yourself (DRY)

EVERY PIECE OF KNOWLEDGE MUST HAVE A SINGLE, UNAMBIGUOUS, AUTHORATIVE REPRESENTATION WITHIN A SYSTEM.

A. Hunt, D. Thomas,
“The Pragmatic Programmer”

How Does Duplication Arise?

- Imposed duplication
- Inadvertent duplication
- Impatient duplication
- Inter-developer duplication

Imposed duplications

- C++ favors compile time verifications
- Everything is declared in advance
- Leads to duplications in certain domains
- Proper DSL eliminates duplication

CPP Macro as a way to develop an embedded DSL when using C++ templates is not an option

Variability Points

Variability Points

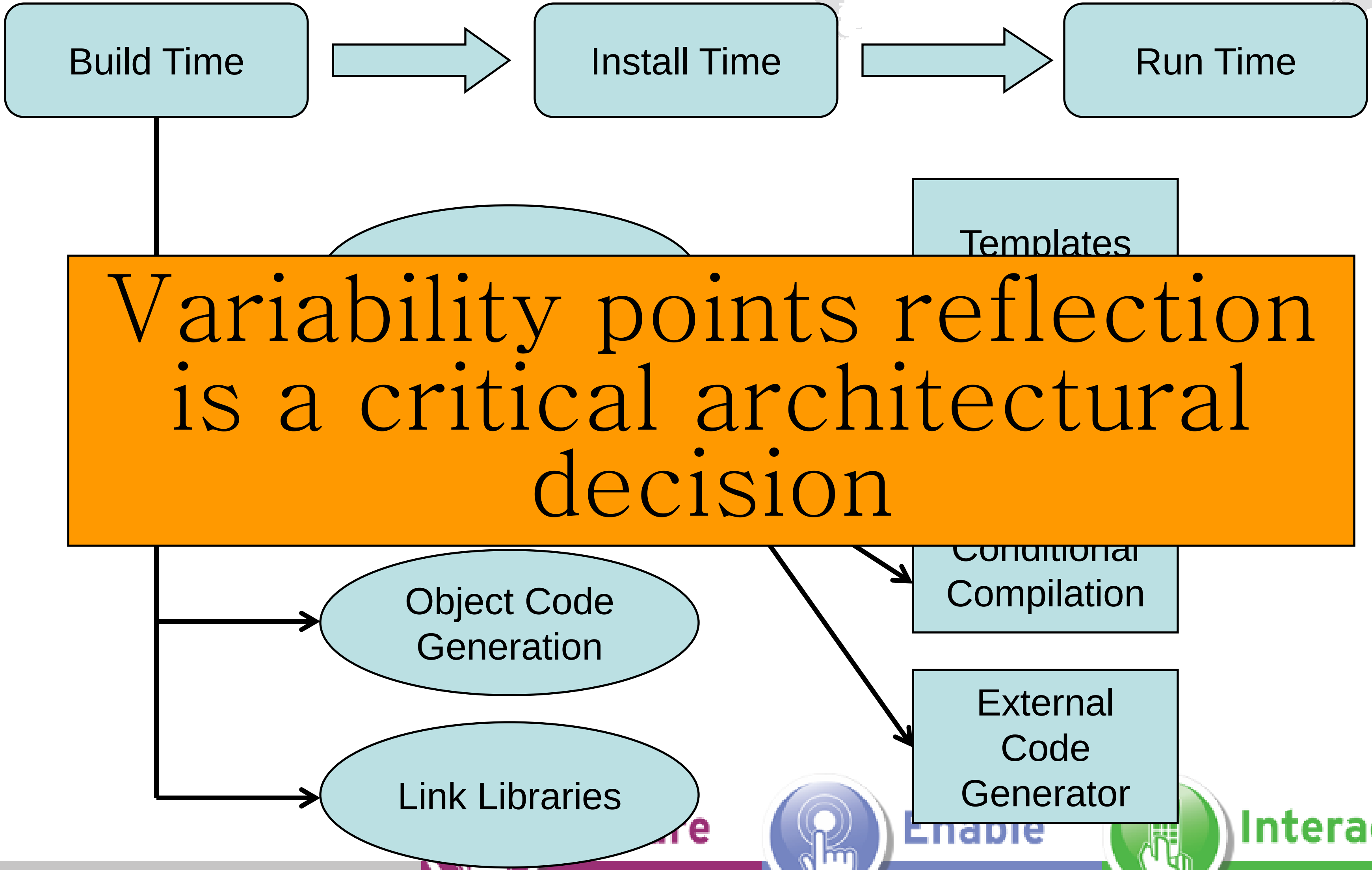
- What could be different:
  - – CPU and Compilers
  - – Hardware Vendors
  - – Platform
  - – Resources
  - – Features

Multi-dimensional Structure

VP1= {$v^{1,1}$, $v^{1,2}$, …}	CPU = {Intel, …}
VP2= {$v^{2,1}$, $v^{2,2}$, …}	NVM = {ST, …}
VP3= {$v^{3,1}$, $v^{3,2}$, …}	Endianess = {BE, LE}
VP4= {$v^{4,1}$, $v^{4,2}$, …}	Target = {Test, Production}
VP5= {$v^{5,1}$, $v^{5,2}$, …}	Debug = {DEBUG, NDEBUG}

Some variability points are mutually dependent (e.g. CPU -> Endianess). In general case variability points could be modeled using a sparse multi-dimensional matrix.

Variability Points Resolution



Variability points reflection is a critical architectural decision

When Macros?

Preprocessor macros are suitable for reflecting variability points through automatic code generation when using C++ templates is not possible and using an external code generator is not desirable.

Secure Enable Interact

Testing Macros

Is it possible at all?

Using PP_ASSERT_EQUAL

```
#include <stdafx.h>
#include <bmock/pp_assert.hpp>
#include <boost/preprocessor/seq/enum.hpp>

namespace
{
    struct pp_assert_tester {};

    #define SEQ (B)(O)(O)(S)(T)

    BMOCK_TEST(pp_assert_tester, test_simple_assertion)
    {
        const char *EXPECTED= "B,O,O,S,T";

        PP_ASSERT_EQUAL( EXPECTED, BOOST_PP_SEQ_ENUM(SEQ) );
    }
}
```

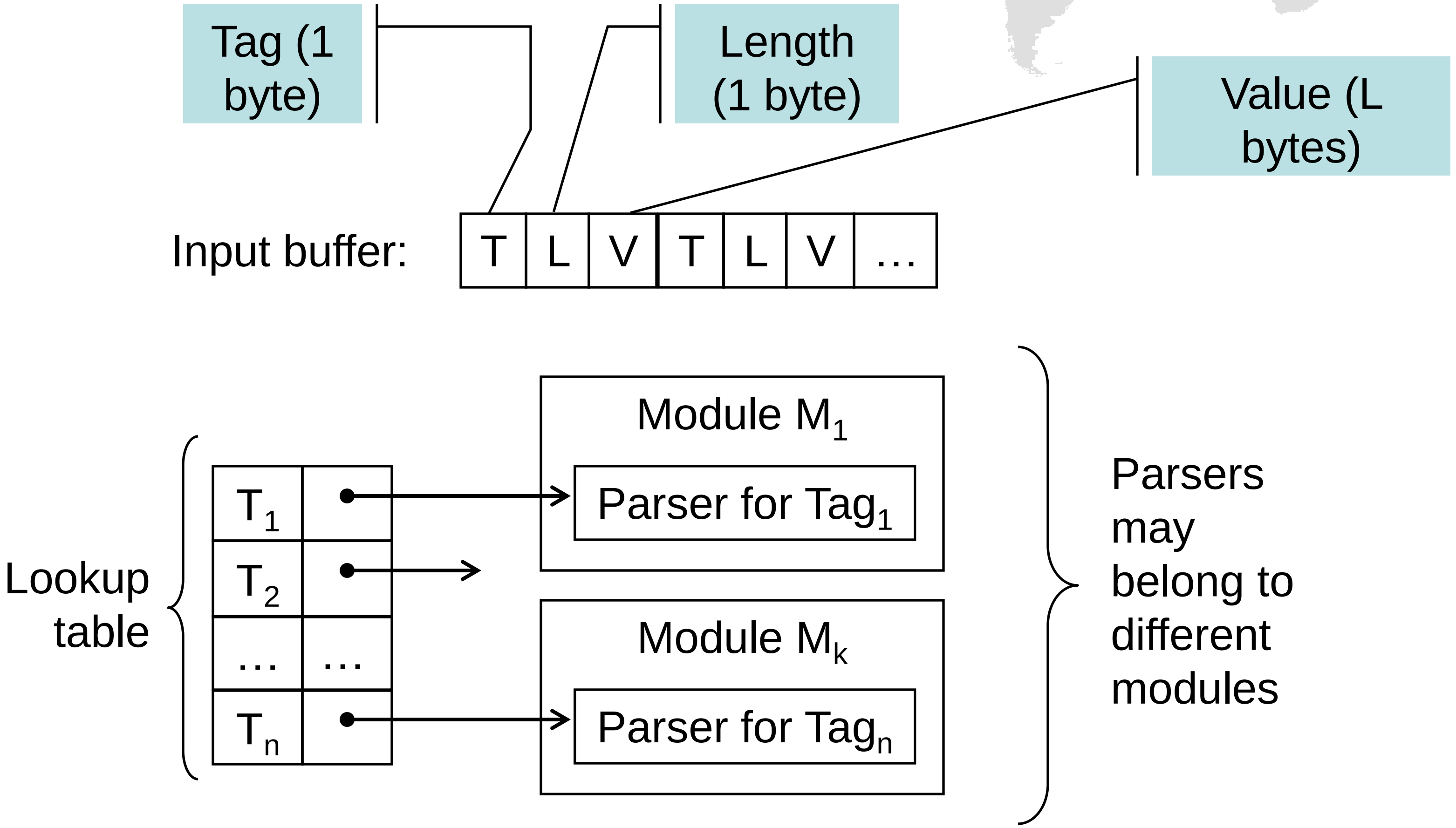
Converts the second argument to string and compares with the first one

Static Data Table Population

Problem Statement

- Need to populate a static data table
- Want to avoid duplicated declarations
- Dynamic containers are not an option due to resource constraints
- Example: a TLV parsers table (e.g. MPEG-2, DVB-SI)

Tag-Length-Value Parsers



Tag (1 byte)

Length (1 byte)

Value (L bytes)

Input buffer:

T	L	V	T	L	V	…

Lookup table

T_1	●
T_2	●
…	…
T_n	●

Module M_1

Parser for Tag_1

Module M_k

Parser for Tag_n

Parsers may belong to different modules

How to reflect this variability point (Tags being supported) at build time without violating DRY?

Any ideas?

“Naïve” Solution

```
#include <stdafx.h>

using namespace std;
typedef unsigned char byte;

struct TlvHandler
{
    byte tag_;
    void (*parser_)(const byte *);
};

struct TlvProcessor
{
    void Process(const byte *begin, const byte *end);

    static const size_t     N = 2;
    static const TlvHandler map_[N];
};
```

Duplication

“Naïve” Solution

```
#include <algorithm>
#include <boost/lambda/bind.hpp>
#include "Module1.h"
#include "Module2.h"
```

Duplication

```
void TlvProcessor::Process(const byte *begin, const byte *end)
{
    namespace bll = boost::lambda;
    while(begin != end)
    {
        const byte tag = *begin++;
        const TlvHandler *pH =
                find_if(map_, map_+N,bind(&TlvHandler::tag_, bll::_1)==tag);
        if (pH != map_+N) pH->parser_(begin);
        begin += *begin + 1;
    }
}

const TlvHandler TlvProcessor::map_[TlvProcessor::N] =
{
    {0x01,Module1::p1},
    {0x02,Module2::p2}
};
```

Duplication

Could we do it better?

Any Ideas?

How about this?

```
#define TlvProcessor_DESCRIPTORS \
        TLV(01,PSI,video_stream_descriptor) \
        TLV(02,PSI,audio_stream_descriptor) \
        TLV(40,SI,network_name_descriptor) \
        TLV(66,SI,data_broadcast_id_descriptor)


DESCRIPTOR_TABLE(TlvProcessor)
```

TlvProcessor

```
struct TlvHandler
{
    byte tag_;
    void (*parser_)(const byte *);
};
struct TlvHandlerMap
{
    const TlvHandler *begi
    {
        return table_;
    }
    const TlvHandler *end(
    {
        return table_ + size_;
    }
    const TlvHandler *table_;
    size_t           size_;
};
struct TlvProcessor
{
    void Process(const byte *begin, const byte *end) const;
    static const TlvHandlerMap map_;
};
```

Object design always should come first (need to know which code to generate)

Using Tuples and Sequences

```
#include <boost/preprocessor/tuple/elem.hpp>

#define TLV(TAG, MODULE, PA
        ((TAG, MODULE, PAR

#define __DESC_TUPLE_SIZE

#define __DESC_TAG(DT)   \
    BOOST_PP_TUPLE_ELEM

#define __DESC_MODULE(DT
    BOOST_PP_TUPLE_ELEM

#define __DESC_PARSER(DT) \
    BOOST_PP_TUPLE_ELEM(__DESC_TUPLE_SIZE, 2, DT)
```

Collections (e.g. sequence) of tuples are the most fundamental PP meta-programming data structures

Automatic Forward Declaration

```
#include <boost/preprocessor/seq/for_each.hpp>

#define __DECLARE_DESCRIPTOR(R, _, DT) \
    namespace __DESC_MODULE(DT)  \
        { extern void __DESC_PARSER(DT) ( const byte *); }

#define __DECLARE_DESCRIPTORS(DESC) \
    BOOST_PP_SEQ_FOR_EACH(__DECLARE_DESCRIPTOR,_,DESC)

BMOCK_TEST(tlv_table_tester, test_declare_descriptors)
{
    const char EXPECTED[] =
        "namespace PSI { extern void video_stream_descriptor(const byte *); }"
        "namespace PSI { extern void audio_stream_descriptor(const byte *); }"
        "namespace SI { extern void network_name_descriptor(const byte *); }"
        "namespace SI { extern void data_broadcast_id_descriptor(const byte *); }"
    ;

    PP_ASSERT_EQUAL( EXPECTED,
        __DECLARE_DESCRIPTORS(TlvProcessor_DESCRIPTORS) );
}
```

Automatic Table Generation

```
#include <boost/preprocessor/seq/for_each_i.hpp>
#include <boost/preprocessor/punctuation/comma_if.hpp>
#include <boost/preprocessor/cat.hpp>

#define __BUILD_HANDLER_ENTRY(R,D,I,DT) \
        BOOST_PP_COMMA_IF(I) \
        {BOOST_PP_CAT(0x,__DESC_TAG(DT)), \
            __DESC_MODULE(DT) :: __DESC_PARSER(DT)}

#define __BUILD_HANDLERS_TABLE(DESC) \
        BOOST_PP_SEQ_FOR_EACH_I(__BUILD_HANDLER_ENTRY,_,DESC)

BMOCK_TEST(tlv_table_tester, test_build_handlers_table)
{
    const char EXPECTED[] =
        "{0x01, PSI :: video_stream_descriptor},"
        "{0x02, PSI :: audio_stream_descriptor},"
        "{0x40, SI :: network_name_descriptor},"
        "{0x66, SI :: data_broadcast_id_descriptor}"
    ;
    PP_ASSERT_EQUAL( EXPECTED,
                __BUILD_HANDLERS_TABLE(TlvProcessor_DESCRIPTORS) );
}
```

Putting ‘Em Together

```
#define __DESCRIPTOR_TABLE(NAME, DESC) \
    __DECLARE_DESCRIPTORS(DESC) \
    const TlvHandler BOOST_PP_CAT(NAME,_Table) [] = \
    { \
        __BUILD_HANDLERS_TABLE(DESC) \
    }; \
    const TlvHandlerMap NAME :: map_ = \
    { \
        BOOST_PP_CAT(NAME,_Table), \
        BOOST_PP_SEQ_SIZE(DESC) \
    }; \

#define DESCRIPTOR_TABLE(NAME) \
    __DESCRIPTOR_TABLE(NAME, BOOST_PP_CAT(NAME,_DESCRIPTORS))
```

Putting ‘Em Together

```
#undef __DECLARE_DESCRIPTORS
#undef __BUILD_HANDLERS_TABLE
#include <boost/preprocessor/stringize.hpp>
#include <boost/preprocessor/facilities/expand.hpp>
#define DESC \
    BOOST_PP_EXPAND(BOOST_PP_STRINGIZE ((TlvProcessor_DESCRIPTORS)))

BMOCK_TEST(tlv_table_tester, test_tlv_table_macro)
{
    const char EXPECTED[] =
        "__DECLARE_DESCRIPTORS" DESC
        "const TlvHandler TlvProcessor_Table[] ="
        "{"
            "__BUILD_HANDLERS_TABLE" DESC
        "};"
        "const TlvHandlerMap TlvProcessor::map_ =
        “{“
                “TlvProcessor_Table,"
                "4“
        “};”
        ;
    PP_ASSERT_EQUAL( EXPECTED, DESCRIPTOR_TABLE(TlvProcessor) );
}
```

Bit Field Manipulation

Secure Enable Interact

Problem Statement

- C/C++ bit fields provide a convenient way for bit level manipulation, however:
  - – Not portable with regard to bid/little endian
  - – Sometimes need to manipulate at byte/word level (e.g. i/o)
  - – Selecting proper base type level violates DRY principle
  - – For some compilers direct shift/mask implementation might be more efficient

How about this?

```
BYTE(
   Flags,
       FLAG(F1)
       FLAG(F2)
       FIELD(F3, 6)
       FIELD(F4, 3)
       FLAG(F5)
) flags_;

input_stream >> flags_;

If (flags_.GetF1())

if (4 == flags_.GetF3())

flags_.SetF4(7);

output_stream << flags_;
```

Defining Sequence Elements

```
#define FIELD(NAME, LENGTH) \
      ((NAME, LENGTH))

#define FLAG(NAME) \
      FIELD(NAME, 1)

BYTE(
   Flags,
      FLAG(F1)
      FLAG(F2)
      FIELD(F3, 6)
      FIELD(F4, 3)
      FLAG(F5)
)
```

Is equivalent to

```
BYTE(Flags, ((F1,1))((F1,1))((F3,6))((F4,3))((F5,1)) )
```

Extracting Tuple Elements

```
#include <boost/preprocessor/tuple/elem.hpp>

#define SEQ FLAG(F1) FLAG(F2) FIELD(F3,6) FIELD(F4,3) FLAG(F5)

#define __FIELD_TUPLE_SIZE 2

#define __FIELD_NAME(FT) \
        BOOST_PP_TUPLE_ELEM(__FIELD_TUPLE_SIZE, 0, FT)

#define __FIELD_LENGTH(FT) \
        BOOST_PP_TUPLE_ELEM(__FIELD_TUPLE_SIZE, 1, FT)
```

Calculate the total size

```
#include <boost/preprocessor/seq/fold_left.hpp>
#include <boost/preprocessor/arithmetic/add.hpp>

#define SEQ FLAG(F1) FLAG(F2) FIELD(F3,6) FIELD(F4,3) FLAG(F5)

#define __ADD_FIELD_LENGTH(R, S, F) \
        BOOST_PP_ADD_D(R, S, __FIELD_LENGTH(F))

#define __CALC_BIT_SIZE(FIELDS) \
        BOOST_PP_SEQ_FOLD_LEFT(__ADD_FIELD_LENGTH, 0, FIELDS)

namespace
{
    struct bit_field_tester{};

    BMOCK_TEST(bit_field_tester, test_get_size)
    {
        const char EXPECTED[] = "12";
        PP_ASSERT_EQUAL( EXPECTED, __CALC_BIT_SIZE(SEQ) );
    }
}
```

Select underlying type

```
namespace
{
    struct bit_field_tester{};

    BMOCK_TEST(bit_field_tester, test_get_type)
    {
        const char EXPECTED[] = "word16";
        PP_ASSERT_EQUAL( EXPECTED, __GET_FIELD_TYPE(SEQ) );
    }
}
```

Select underlying type

```
#include <boost/preprocessor/control/if.hpp>
#include <boost/preprocessor/comparison/less.hpp>
#include <boost/preprocessor/assert_msg.hpp>

typedef unsigned char  byte;
typedef unsigned short word16;
typedef unsigned long  word32;

#define __UNSUPPORTED_FIELD_LENGTH \
        BOOST_PP_ASSERT_MSG(0,"Unsupported bit field struc length (>32)") \

#define __GET_FIELD_TYPE2(SIZE) \
        BOOST_PP_IF(BOOST_PP_LESS(SIZE, 9), byte, \
        BOOST_PP_IF(BOOST_PP_LESS(SIZE, 17), word16, \
        BOOST_PP_IF(BOOST_PP_LESS(SIZE, 33), word32, \
        __UNSUPPORTED_FIELD_LENGTH \
        )))

#define __GET_FIELD_TYPE(FIELDS) \
        __GET_FIELD_TYPE2(__CALC_BIT_SIZE(FIELDS))
```

Generate Assessors

```
BMOCK_TEST(bit_field_tester, test_generate_accessors)
{
    const char EXPECTED[] =
        "bool GetF1() const { return as_bits_.F1_; }"
        "void SetF1(bool v) { as_bits_.F1_ = v ? 1 : 0; }"
        "bool GetF2() const { return as_bits_.F2_; }"
        "void SetF2(bool v) { as_bits_.F2_ = v ? 1 : 0; }"
        "int GetF3() const { return as_bits_.F3_; }"
        "void SetF3(int v) { as_bits_.F3_ = v; }"
        "int GetF4() const { return as_bits_.F4_; }"
        "void SetF4(int v) { as_bits_.F4_ = v; }"
        "bool GetF5() const { return as_bits_.F5_; }"
        "void SetF5(bool v) { as_bits_.F5_ = v ? 1 : 0; }"
        ;

    PP_ASSERT_EQUAL( EXPECTED, __GENERATE_ACCESSORS(SEQ) );
}
```

Generate Assessors

```
#include <boost/preprocessor/seq/for_each.hpp>
#include <boost/preprocessor/cat.hpp>

#define __FIELD_TYPE(FT) \
        BOOST_PP_IF(BOOST_PP_LESS(1, __FIELD_LENGTH(FT)), int ,bool)

#define __BIT_FIELD_NAME(N)  BOOST_PP_CAT(N,_)

#define __FIELD_ASSIGN(L, V)  BOOST_PP_IF(BOOST_PP_LESS(1, L), V, V ? 1 : 0)

#define __DEFINE_GET_ACCESSOR(T, N) \
        T BOOST_PP_CAT(Get,N) () const { return as_bits_ . __BIT_FIELD_NAME(N) ; }

#define __DEFINE_SET_ACCESSOR(T, N, L) \
        void BOOST_PP_CAT(Set,N) (T v) { as_bits_ . __BIT_FIELD_NAME(N) = \
                __FIELD_ASSIGN(L, v); }

#define __DEFINE_FIELD_ACCESSORS(R, D, FT) \
        __DEFINE_GET_ACCESSOR(__FIELD_TYPE(FT), __FIELD_NAME(FT)) \
        __DEFINE_SET_ACCESSOR(__FIELD_TYPE(FT), __FIELD_NAME(FT), \
                __FIELD_LENGTH(FT))

#define __GENERATE_ACCESSORS(FIELDS) \
        BOOST_PP_SEQ_FOR_EACH(__DEFINE_FIELD_ACCESSORS, _, FIELDS)
```

Generate Bit Fields

```
BMOCK_TEST(bit_field_tester, test_generate_bit_fields)
{
    const char EXPECTED[] =
        "struct {"
            "word16 _    : 4;“
            "word16 F5_ : 1;"
            "word16 F4_ : 3;"
            "word16 F3_ : 6;"
            "word16 F2_ : 1;"
            "word16 F1_ : 1;"
        "} as_bits_;"
    ;
    PP_ASSERT_EQUAL( EXPECTED, __GENERATE_BIT_FIELDS(SEQ) );
}
```

Spare bits field

Reflects little endian bits order

Generate Bit Fields

```
#include <boost/preprocessor/arithmetic/sub.hpp>
#include <boost/preprocessor/seq/push_back.hpp>
#include <boost/preprocessor/seq/fold_right.hpp>

#define __GET_SPARE_BITS(SIZE) \
        BOOST_PP_IF(BOOST_PP_LESS(SIZE, 9), BOOST_PP_SUB(8, SIZE), \
        BOOST_PP_IF(BOOST_PP_LESS(SIZE, 17), BOOST_PP_SUB(16, SIZE), \
        BOOST_PP_IF(BOOST_PP_LESS(SIZE, 33), BOOS_PP_SUB(32, SIZE), \
        __UNSUPPORTED_FIELD_LENGTH \
        )))

#define __ADD_SPARE_BITS(SPARE_SIZE, FIELDS) \
        BOOST_PP_IF(SPARE_SIZE, \
                BOOST_PP_SEQ_PUSH_BACK(FIELDS,(,SPARE_SIZE)), FIELDS)

#define __DEFINE_FIELD(R, TYPE, FT) \
        TYPE __BIT_FIELD_NAME(__FIELD_NAME(FT)) : __FIELD_LENGTH(FT);
```

Generate Bit Fields

```
#define __ADD_FIELD_TO_STATE(S, STATE, FT) \
    ( \
        BOOST_PP_TUPLE_ELEM(2,0,STATE) \
            __DEFINE_FIELD(_, BOOST_PP_TUPLE_ELEM(2,1,STATE), FT) , \
        BOOST_PP_TUPLE_ELEM(2,1,STATE) \
    ) \

#ifdef BIG_ENDIAN
#define __GENERATE_BIT_FIELDS2(TYPE, FIELDS) \
    BOOST_PP_SEQ_FOR_EACH(__DEFINE_FIELD,TYPE,FIELDS)
#else
#define __GENERATE_BIT_FIELDS2(TYPE, FIELDS) \
    BOOST_PP_TUPLE_ELEM(2,0, \
        BOOST_PP_SEQ_FOLD_RIGHT(__ADD_FIELD_TO_STATE, \
            (,TYPE),FIELDS) \
    )
#endif
#define __GENERATE_BIT_FIELDS(FIELDS) \
    struct \
    { \
        __GENERATE_BIT_FIELDS2(__GET_FIELD_TYPE(FIELDS), \
            __ADD_SPARE_BITS(__GET_SPARE_BITS(__CALC_BIT_SIZE(FIELDS)), \
                FIELDS)) \
    } as_bits_; \
```

Too much duplication!

Need to refactor and to bind them all together

Byte Macro

```
#undef __GENERATE_ACCESSORS
#undef __GENERATE_BIT_FIELDS

BMOCK_TEST(bit_field_tester, test_byte_macro)
{
	const char EXPECTED[] =
		"union Flags {"
			"__GENERATE_ACCESSORS(((F1,1))((F2,1))((F3,6))((F4,3))((F5,1)))"
			"word16 as_word16_;"
			"__GENERATE_BIT_FIELDS(word16, 4, ((F1,1))((F2,1))((F3,6))((F4,3))((F5,1)));"
		"}"
		;
	PP_ASSERT_EQUAL( EXPECTED, BYTE(Flags, SEQ) );
}
```

Programming by intentions

Need to define type

Need to calculate spare bits

Byte Macro

```
BMOCK_TEST(bit_field_tester, test_get_byte_size)
{
	PP_ASSERT_EQUAL( "8", __GET_BYTE_SIZE(7) );
	PP_ASSERT_EQUAL( "16", __GET_BYTE_SIZE(12) );
	PP_ASSERT_EQUAL( "32", __GET_BYTE_SIZE(21) );
}

BMOCK_TEST(bit_field_tester, test_get_field_type)
{
	PP_ASSERT_EQUAL( "byte", __GET_FIELD_TYPE(8) );
	PP_ASSERT_EQUAL( "word16", __GET_FIELD_TYPE(16) );
	PP_ASSERT_EQUAL( "word32", __GET_FIELD_TYPE(32) );
}
```

Byte Macro

```
#define FIELD(NAME, LENGTH) \
      ((NAME, LENGTH))

#define FLAG(NAME) \
      FIELD(NAME, 1)

#define __FIELD_TUPLE_SIZE 2

#define __FIELD_NAME(FT) \
      BOOST_PP_TUPLE_ELEM(__FIELD_TUPLE_SIZE, 0, FT)

#define __FIELD_LENGTH(FT) \
      BOOST_PP_TUPLE_ELEM(__FIELD_TUPLE_SIZE, 1, FT)

#define __FIELD_TYPE(FT) \
      BOOST_PP_IF(BOOST_PP_LESS(1, __FIELD_LENGTH(FT)), int ,bool)

#define __BIT_FIELD_NAME(N) \
      BOOST_PP_CAT(N,_)

#define __ADD_FIELD_LENGTH(R, S, F) \
      BOOST_PP_ADD_D(R, S, __FIELD_LENGTH(F))

#define __CALC_BIT_SIZE(FIELDS) \
      BOOST_PP_SEQ_FOLD_LEFT(__ADD_FIELD_LENGTH, 0, FIELDS)
```

Byte Macro

```
typedef unsigned char  byte;
typedef unsigned short word16;
typedef unsigned long  word32;

#define __UNSUPPORTED_FIELD_LENGTH \
        BOOST_PP_ASSERT_MSG(0,"Unsupported bit field struct length (>32)") \

#define __TYPED_NAME(TYPE) \
        BOOST_PP_CAT(BOOST_PP_CAT(as_,TYPE),_) \

#define __FIELD_TYPE_8  byte
#define __FIELD_TYPE_16 word16
#define __FIELD_TYPE_32 word32

#define __GET_FIELD_TYPE(BYTE_SIZE) \
        BOOST_PP_CAT(__FIELD_TYPE_, BYTE_SIZE)

#define __GET_BYTE_SIZE(BIT_SIZE) \
        BOOST_PP_IF(BOOST_PP_LESS(BIT_SIZE, 9) , 8, \
        BOOST_PP_IF(BOOST_PP_LESS(BIT_SIZE, 17), 16, \
        BOOST_PP_IF(BOOST_PP_LESS(BIT_SIZE, 33), 32, \
        __UNSUPPORTED_FIELD_LENGTH \
        )))
```

Byte Macro

```
#define __DEFINE_FIELD(R, TYPE, FT) \
        TYPE __BIT_FIELD_NAME(__FIELD_NAME(FT)) : __FIELD_LENGTH(FT);

#define __ADD_FIELD_TO_STATE(S, STATE, FT) \
    ( \
        BOOST_PP_TUPLE_ELEM(2,0,STATE)  \
            __DEFINE_FIELD(_, BOOST_PP_TUPLE_ELEM(2,1,STATE), FT)  \
        ,BOOST_PP_TUPLE_ELEM(2,1,STATE) \
    ) \

#ifdef BIG_ENDIAN
#define __GENERATE_BIT_FIELDS2(TYPE, FIELDS) \
    BOOST_PP_SEQ_FOR_EACH(__DEFINE_FIELD,TYPE,FIELDS)
#else
#define __GENERATE_BIT_FIELDS2(TYPE, FIELDS) \
    BOOST_PP_TUPLE_ELEM(2,0, \
        BOOST_PP_SEQ_FOLD_RIGHT(__ADD_FIELD_TO_STATE,(,TYPE),FIELDS) \
    )
#endif

#define __ADD_SPARE_BITS(SPARE, FIELDS) \
    BOOST_PP_IF(SPARE, BOOST_PP_SEQ_PUSH_BACK(FIELDS,(,SPARE)), FIELDS)
```

Byte Macro

```
#define __GENERATE_BIT_FIELDS(TYPE, SPARE, FIELDS) \
    struct \
    { \
        __GENERATE_BIT_FIELDS2(TYPE,__ADD_SPARE_BITS(SPARE, FIELDS)) \
    } as_bits_; \

#define __FIELD_ASSIGN(L, V) \
    BOOST_PP_IF(BOOST_PP_LESS(1, L), V, V ? 1 : 0) \

#define __DEFINE_GET_ACCESSOR(T, N) \
    T BOOST_PP_CAT(Get,N) () const { return as_bits_ . __BIT_FIELD_NAME(N) ; }

#define __DEFINE_SET_ACCESSOR(T, N, L) \
    void BOOST_PP_CAT(Set,N) (T v) { as_bits_ . __BIT_FIELD_NAME(N) = \
        __FIELD_ASSIGN(L, v); }

#define __DEFINE_FIELD_ACCESSORS(R, D, FT) \
    __DEFINE_GET_ACCESSOR(__FIELD_TYPE(FT), __FIELD_NAME(FT)) \
    __DEFINE_SET_ACCESSOR(__FIELD_TYPE(FT), __FIELD_NAME(FT), \
        __FIELD_LENGTH(FT))

#define __GENERATE_ACCESSORS(FIELDS) \
        BOOST_PP_SEQ_FOR_EACH(__DEFINE_FIELD_ACCESSORS, _, FIELDS)
```

Byte Macro

```
#define __BYTE3(NAME, TYPE, SPARE, FIELDS) \
    union NAME { \
        __GENERATE_ACCESSORS(FIELDS) \
        TYPE __TYPED_NAME(TYPE); \
        __GENERATE_BIT_FIELDS(TYPE, SPARE, FIELDS); \
    } \

#define __BYTE2(NAME, BYTE_SIZE, BIT_SIZE, FIELDS) \
    __BYTE3(NAME, __GET_FIELD_TYPE(BYTE_SIZE),  \
        BOOST_PP_SUB(BYTE_SIZE, BIT_SIZE), FIELDS)

#define __BYTE(NAME, BIT_SIZE, FIELDS) \
    __BYTE2(NAME, __GET_BYTE_SIZE(BIT_SIZE), BIT_SIZE, FIELDS) \

#define BYTE(NAME, FIELDS) \
    __BYTE(NAME, __CALC_BIT_SIZE(FIELDS), FIELDS)
```

Final Test

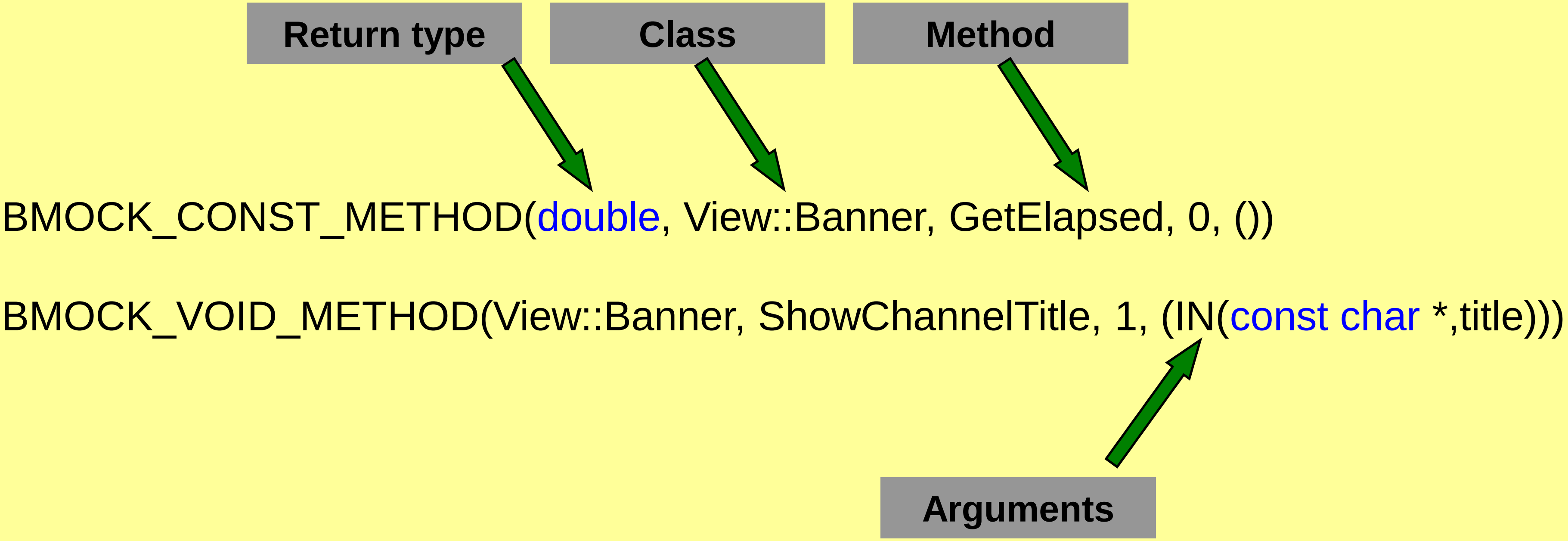
```
BMOCK_TEST(bit_field_tester, test_byte_16)
{
	BYTE(Flags, SEQ) flags = {0x8F80};

	BOOST_CHECK(flags.GetF1());
	BOOST_CHECK(!flags.GetF2());
	BOOST_CHECK_EQUAL(15,flags.GetF3());
	BOOST_CHECK_EQUAL(4,flags.GetF4());
	BOOST_CHECK(!flags.GetF5());
	flags.SetF1(false);
	flags.SetF2(true);
	flags.SetF3(0);
	flags.SetF4(7);
	flags.SetF5(true);
	BOOST_CHECK_EQUAL(0x40F0, flags.as_word16_);
}
```

IDL-like Annotations

Problem Statement

- Want to use an IDL-like annotations of functions and methods to automatically generate:
  - – Static mocks
  - – Dynamic mocks
  - – Console i/o adapters
  - – Python adapters
  - – Tracing adapters
  - – Profiling adapters

Examples



Return type

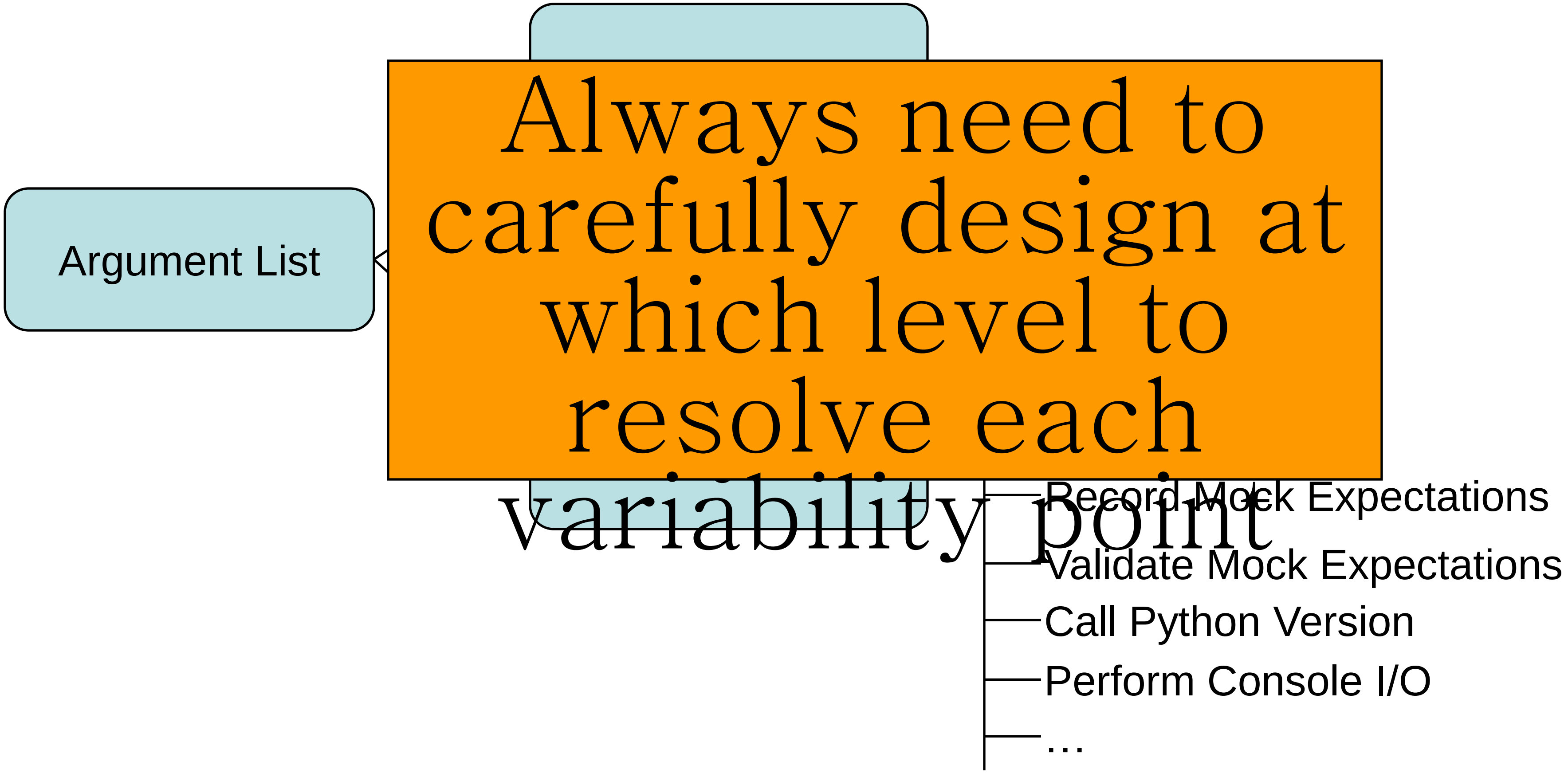
Class

Method

```
BMOCK_CONST_METHOD(double, View::Banner, GetElapsed, 0, ())

BMOCK_VOID_METHOD(View::Banner, ShowChannelTitle, 1, (IN(const char *,title)))
```

Arguments

Argument List Processing



Always need to carefully design at which level to resolve each variability point

Unification

```
#include <boost/preprocessor/tuple/to_seq.hpp>

#define BMOCK_CONST_VOID_METHOD(CN, MN, AN, AL) \
        __BMOCK_FUNCTION(_VOID, \
        void, \
        CN :: MN, \
        BOOST_PP_TUPLE_TO_SEQ(AN, AL), \
        const)
```

Means “Has no return”

Return type

Convert tuple to sequence for further processing

Full function name

Const specification

Function Declaration

```
#define __BMOCK_FUNCTION(HAS_RETURN, RT, FN, ARGS, CONST) \
          FT FN (__BMOCK_PROCESS_ARGS(DECL_, ARGS)) CONST \
          ….
          ….
```

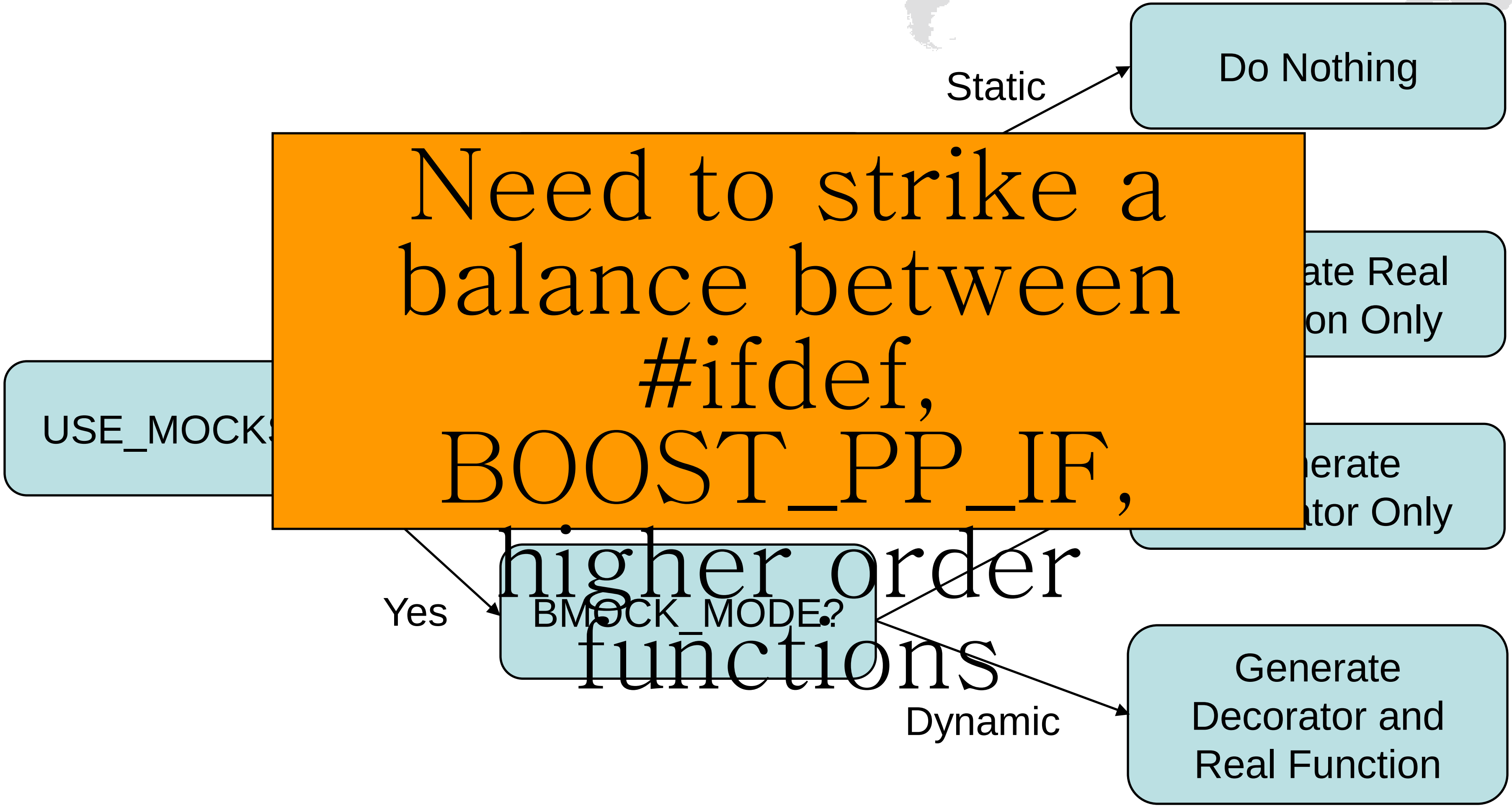
Processing Argument List

```
#include <boost/preprocessor/facilities/expand.hpp>
#include <boost/preprocessor/cat.hpp>
#include <boost/preprocessor/seq/for_each_i.hpp>
#include <boost/preprocessor/punctuation/comma_if.hpp>

#define __BMOCK_PROCESS_ARG(R, PREFIX, I,ARG) \
    BOOST_PP_COMMA_IF(I) \
    BOOST_PP_EXPAND( \
        BOOST_PP_CAT(BOOST_PP_CAT(__BMOCK_,PREFIX),ARG)
    )

#define __BMOCK_PROCESS_ARGS(PREFIX, ARGS) \
    BOOST_PP_SEQ_FOR_EACH_I(
        __BMOCK_PROCESS_ARG, \
        PREFIX, \
        ARGS\
    )
```

Processing One Argument

```
#define __BMOCK_DECL_IN(TYPE, NAME) TYPE NAME
#define __BMOCK_CALL_IN(TYPE, NAME) NAME
//
// Other, more sophisticated, argument processing types will be defined here
//
```

BMock Decision Tree



Need to strike a balance between #ifdef, BOOST_PP_IF, higher order functions

Dispatching Decorator Type

Normally set once per project

Defined on per mock basis

```
#ifdef BMOCK_USE_MOCKS
#define __BMOCK_FUNCTION(IS_VOID, RT, FN, ARGS, CONST) \
      __BMOCK_APPLY(GENERATE_,IS_VOID,RT,FN,ARGS,CONST)
#else
#define __BMOCK_FUNCTION(IS_VOID, RT, FN, ARGS, CONST) \
      __BMOCK_APPLY(SKIP_,IS_VOID,RT,FN,ARGS,CONST)
#endif
```

Dispatching Decorator Type

```
#define __BMOCK_APPLY(VERB, IS_VOID, RT, FN, ARGS, CONST) \
    BOOST_PP_CAT( \
        BOOST_PP_CAT( \
            BOOST_PP_CAT(__BMOCK_,VERB), \
            BOOST_PP_CAT(BMOCK_MODE, IS_VOID) \
        ), \
        _MOCK \
    ) (HR,RT,FN,ARGS,CONST)
```

Normally has a default value per project, but could be overridden for particular compilation unit

Dispatching Decorator Type

```
#define __BMOCK_DECLARE_FUNCTION(RT, FN, ARGS, CONST) \
          RT FN (__BMOCK_PROCESS_ARGS(DECL_, ARGS)) CONST

#define __BMOCK_GENERATE_DYNAMIC_MOCK(RT, FN, ARGS, CONST) \
     __BMOCK_DECLARE_FUNCTION(RT, FN, ARGS, CONST) \
     ...
#define __BMOCK_GENERATE_STATIC_MOCK(RT, FN, ARGS, CONST) \
     __BMOCK_DECLARE_FUNCTION(RT, FN, ARGS, CONST) \
     ...
#define __BMOCK_SKIP_DYNAMIC_MOCK(RT, FN, ARGS, CONST) \
     __BMOCK_DECLARE_FUNCTION(RT, FN, ARGS, CONST) \
     ...
#define __BMOCK_SKIP_STATIC_MOCK(RT, FN, ARGS, CONST)

#define __BMOCK_GENERATE_DYNAMIC_VOID_MOCK(RT, FN, ARGS, CONST) \
     __BMOCK_DECLARE_FUNCTION(RT, FN, ARGS, CONST) \
     ...
#define __BMOCK_GENERATE_STATIC_VOID_MOCK(RT, FN, ARGS, CONST) \
     __BMOCK_DECLARE_FUNCTION(RT, FN, ARGS, CONST) \
     ...
#define __BMOCK_SKIP_DYNAMIC_VOID_MOCK(RT, FN, ARGS, CONST) \
     __BMOCK_DECLARE_FUNCTION(RT, FN, ARGS, CONST) \
     ...
#define __BMOCK_SKIP_STATIC_VOID_MOCK(RT, FN, ARGS, CONST)
```

Summary

- Use macro for embedded DSLs
- Variability points are resolved by architecture
- Macros resolve variability points through code generation
- Always start with object design
- Use collections of tuples to define your DSL
- Strike a balance between #ifdef, BOOST_PP_IF, and higher order functions

Future Directions

- Integration with Wave library
- Supporting fully fledged script (Python?)
- Using macro for external DSLs
- Drop me a line if you any have any comments/ideas:

asher.sterkin@gmail.com

asterkin@nds.com

Secure Enable Interact