

Implementing Type-Driven Wire Protocols using Boost Fusion

Thomas Rodgers | Sr. Software Engineer
DRW Trading Group

Why not ProtoBufs, Thrift, etc?

- These solutions are great when you control *both* ends of the communication
 - Also when cross language compatibility is required
- Many important use-cases where this is not the case
 - 3rd party systems
 - Embedded devices
 - Legacy systems
 - Financial markets

Why not “good ‘ol packed structs”?

- The classic “C” way
- Easy, efficient
 - A simple `reinterpret_cast<>` away from goodness
- Except for that pesky network/host order thing
- Many fundamental protocols implemented this way (TCP, IP, etc.)

Why not “good ‘ol packed structs”?

- Limited abstraction capabilities
 - POD types
 - Forces the third party’s type declarations into *your* domain
- Still have to do member-wise fix-ups for endianness
 - Error prone, maintenance issue
- Quickly falls apart when you have more than one variable length data member
- Difficult to reuse implementation

Reflection?

- If there were a way to do member-wise iteration and type deduction, fairly straightforward to write arbitrary codecs
- Nothing available in Standard C++ today
 - There is an active Standards Committee study group SG7
 - Initial focus on compile time reflection

What about Boost Fusion?

From the Boost Fusion documentation —

Fusion is a library and a framework similar to both STL and the boost MPL. The structure is modeled after MPL, which is modeled after STL. It is named "fusion" because the library is reminiscent of the "fusion" of compile time meta-programming with runtime programming. The library inherently has some interesting flavors and characteristics of both MPL and STL. It lives in the twilight zone between compile time meta-programming and run time programming. STL containers work on values. MPL containers work on types. Fusion containers work on both types and values.

What about Boost Fusion?

From the Boost Fusion documentation —

Fusion is a library and a framework similar to both STL and the boost MPL. The structure is modeled after MPL, which is modeled after STL. It is named "fusion" because the library is reminiscent of the "fusion" of compile time meta-programming with runtime programming. The library inherently has some interesting flavors and characteristics of both MPL and STL. It lives in the twilight zone between *compile time meta-programming* and run time programming. STL containers work on values. MPL containers work on types. Fusion containers work on both types and values.

What about Boost Fusion?

From the Boost Fusion documentation —

Fusion is a library and a framework similar to both STL and the boost MPL. The structure is modeled after MPL, which is modeled after STL. It is named "fusion" because the library is reminiscent of the "fusion" of compile time meta-programming with runtime programming. The library inherently has some interesting flavors and characteristics of both MPL and STL. It lives in the twilight zone between compile time meta-programming and *run time programming*. STL containers work on values. MPL containers work on types. Fusion containers work on both types and values.

What about Boost Fusion?

From the Boost Fusion documentation —

Fusion is a library and a framework similar to both STL and the boost MPL. The structure is modeled after MPL, which is modeled after STL. It is named "fusion" because the library is reminiscent of the "fusion" of compile time meta-programming with runtime programming. The library inherently has some interesting flavors and characteristics of both MPL and STL. It lives in the twilight zone between compile time meta-programming and run time programming. *STL containers work on **values***. MPL containers work on types. Fusion containers work on both types and values.

What about Boost Fusion?

From the Boost Fusion documentation —

Fusion is a library and a framework similar to both STL and the boost MPL. The structure is modeled after MPL, which is modeled after STL. It is named "fusion" because the library is reminiscent of the "fusion" of compile time meta-programming with runtime programming. The library inherently has some interesting flavors and characteristics of both MPL and STL. It lives in the twilight zone between compile time meta-programming and run time programming. STL containers work on values. *MPL containers work on **types**.* Fusion containers work on both types and values.

What about Boost Fusion?

From the Boost Fusion documentation —

Fusion is a library and a framework similar to both STL and the boost MPL. The structure is modeled after MPL, which is modeled after STL. It is named "fusion" because the library is reminiscent of the "fusion" of compile time meta-programming with runtime programming. The library inherently has some interesting flavors and characteristics of both MPL and STL. It lives in the twilight zone between compile time meta-programming and run time programming. STL containers work on values. MPL containers work on types. *Fusion containers work on both **types and values**.*

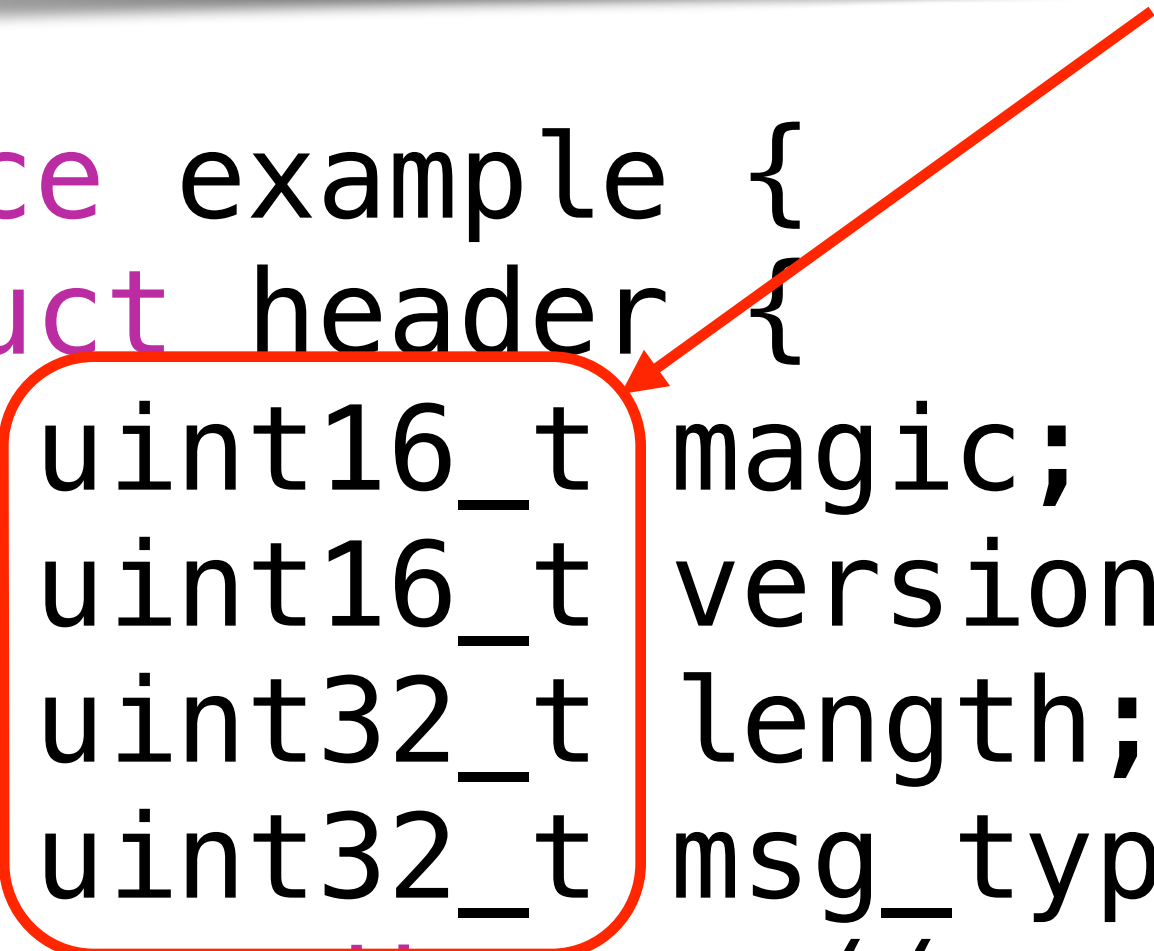
A container of types and values

```
namespace example {  
    struct header {  
        uint16_t magic;  
        uint16_t version;  
        uint32_t length;  
        uint32_t msg_type;  
    } __attribute__((packed));  
}
```

A container of types and values

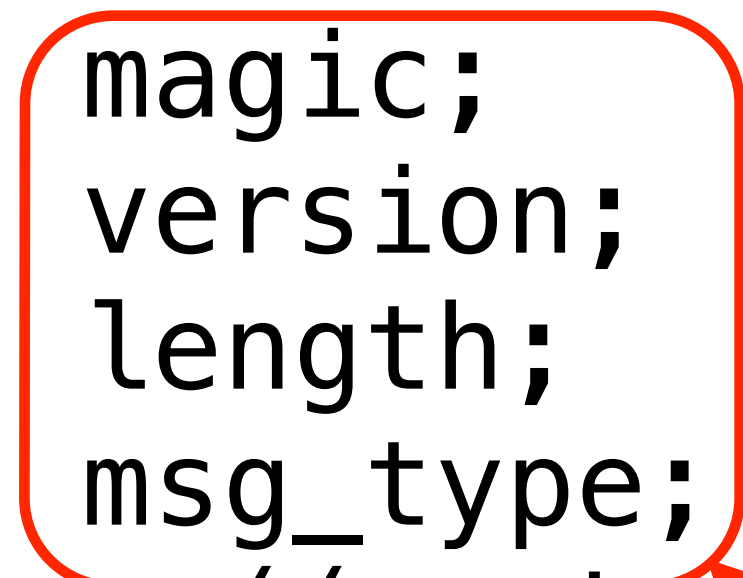
```
mpl::list<uint16_t, uint16_t, uint32_t, uint16_t>
```

```
namespace example {  
    struct header {  
        uint16_t magic;  
        uint16_t version;  
        uint32_t length;  
        uint32_t msg_type;  
    } __attribute__((packed));  
}
```



A container of types and values

```
namespace example {  
    struct header {  
        uint16_t magic;  
        uint16_t version;  
        uint32_t length;  
        uint32_t msg_type;  
    } __attribute__((packed));  
}
```



```
std::tuple<uint16_t&, uint16_t&, uint32_t&, uint16_t&>
```

From packed struct to Fusion struct

```
#include <cstdint>

namespace example {
    struct header {
        uint16_t magic;
        uint16_t version;
        uint32_t length;
        uint32_t msg_type;
    } __attribute__((packed));
}
```

From packed struct to Fusion struct

```
#include <cstdint>
#include <boost/fusion/include/define_struct.hpp>

BOOST_FUSION_DEFINE_STRUCT(
    (example), header,
    (uint16_t, magic)
    (uint16_t, version)
    (uint32_t, length)
    (uint32_t, msg_type)
)
```


Asio buffers

- Two flavors
 - `const_buffer`
 - `mutable_buffer`
- Does not own underlying storage
 - Holds a pointer and a length
- Supports `operator+(size_t n)`
- Free functions
 - `buffer_cast<T*>`
 - `buffer_size`
 - `buffer_copy`

Member-wise visitation

```
#include <boost/fusion/include/for_each.hpp>
```

```
#include <boost/asio/buffer.hpp>
```

```
struct reader {
```

```
    // ...
```

```
};
```

```
template<typename T>
```

```
T read(asio::const_buffer b) {
```

```
    reader r(std::move(b));
```

```
    T res;
```

```
    fusion::for_each(res, r);
```

```
    return res;
```

```
}
```

```
}
```

Member-wise visitation

```
#include <boost/fusion/include/for_each.hpp>
```

```
#include <boost/asio/buffer.hpp>
```

```
struct reader {  
    asio::const_buffer buf_;
```

```
    explicit reader(asio::const_buffer buf)  
        : buf_(std::move(buf))  
    { }
```

```
    template<class T>  
    void operator()(T & val) {  
        // ...  
    }
```

```
};
```

Member-wise visitation

```
#include <boost/fusion/include/for_each.hpp>
```

```
#include <boost/asio/buffer.hpp>
```

```
struct reader {  
    mutable asio::const_buffer buf_;  
  
    explicit reader(asio::const_buffer buf)  
        : buf_(std::move(buf))  
    { }
```

```
    template<class T>  
    void operator()(T & val) const {  
        // ...  
    }  
};
```

Member-wise visitation

```
#include <boost/asio/buffer.hpp>
#include <boost/fusion/include/for_each.hpp>

struct writer {
    mutable asio::mutable_buffer buf_;

    explicit writer(asio::mutable_buffer buf)
    : buf_(std::move(buf))
    { }

    template<class T>
    void operator()(T const& val) const {
        // ...
    }
};

template<typename T>
asio::mutable_buffer write(asio::mutable_buffer b, T const& val) {
    writer w(std::move(b));
    boost::fusion::for_each(res, w);
    return w.buf_;
}
```

Fixing up byte ordering

- Network protocols typically require member-wise fix-ups for endianness
 - ntohl, ntohs, htonl, htons, bswap etc.
 - proposal to add generic ntohs/hton for unsigned integral types to Standard C++
 - Fairly easy to roll our own generic ntohs/hton

Fixing up byte ordering

```
template<class T>
T ntoh(T val) {
    // ...
}

struct reader {
    asio::const_buffer buf_;

    // ...

    template<class T>
    void operator()(T & val) const {
        val = ntoh(*asio::buffer_cast<T const*>(buf_));
        buf_ = buf_ + sizeof(T);
    }
};
```

Fixing up byte ordering

```
template<class T>
T hton(T val) {
    // ...
}

struct writer {
    mutable asio::mutable_buffer buf_;

    // ...

    template<class T>
    void operator()(T const& val) const {
        *asio::buffer_cast<T*>(buf_) = hton(val);
        buf_ = buf + sizeof(T);
    }
};
```


Enumerated Values

```
BOOST_FUSION_DEFINE_STRUCT(  
    (example), header,  
    (example::magic_t, magic)  
    (example::version_t, version)  
    (uint32_t, length)  
    (uint32_t, msg_type)  
)
```

Enumerated Values

```
namespace example {  
    enum class msg_type_t : uint32_t {  
        // ...  
    };  
}
```

```
BOOST_FUSION_DEFINE_STRUCT(  
    (example), header,  
    (example::magic_t, magic)  
    (example::version_t, version)  
    (uint32_t, length)  
    (example::msg_type_t, msg_type)  
)
```

Enumerated Values

```
struct reader {  
    // ...  
  
    template<class T>  
    void operator()(T & val) const {  
        val = ntohs(*asio::buffer_cast<T const*>(buf_));  
        buf_ = buf_ + sizeof(T);  
    }  
};
```

Enumerated Values

```
struct reader {  
    // ...  
  
    template<class T>  
    auto operator()(T & val) const ->  
        typename std::enable_if<std::is_integral<T>::value>::type {  
        val = ntohs(*asio::buffer_cast<T const*>(buf_));  
        buf_ = buf_ + sizeof(T);  
    }  
};
```

Enumerated Values

```
struct reader {  
    // ...  
  
    template<class T>  
    auto operator()(T & val) const ->  
        typename std::enable_if<std::is_enum<T>::value>::type {  
  
        typename std::underlying_type<T>::type v;  
        (*this)(v);  
        val = static_cast<T>(v);    }  
};
```

Fixed “tag” data

- Many protocols have fixed “tags”
 - *magic* signature bytes
 - protocol version markers
- We don’t really care what these are, we just want to encode type and expected value
 - Sounds a lot like `std::integral_constant<>`

Fixed “tag” data

```
#include <cstdint>
#include <boost/fusion/include/define_struct.hpp>

BOOST_FUSION_DEFINE_STRUCT(
    (example), header,
    (uint16_t, magic)
    (uint16_t, version)
    (uint32_t, length)
    (uint32_t, msg_type)
)
```

Fixed “tag” data

```
#include <cstdint>
#include <type_traits>
#include <boost/fusion/include/define_struct.hpp>

namespace example {
    using magic_t = integral_constant<uint16_t, 0xf00d>;
    using version_t = integral_constant<uint16_t, 0xbeef>;
}

BOOST_FUSION_DEFINE_STRUCT(
    (example), header,
    (example::magic_t, magic)
    (example::version_t, version)
    (uint32_t, length)
    (uint32_t, msg_type)
)
```


Fixed “tag” data

```
struct reader {  
    // ...  
  
    template<class T, T v>  
    void operator()(integral_constant<T, v>) const {  
        typedef integral_constant<T, v> type;  
        typename type::value_type val;  
        (*this)(val);  
        if (val != type::value)  
            throw ...;  
    }  
};
```

Fixed “tag” data

```
struct writer {  
    // ...  
  
    template<class T, T v>  
    void operator()(std::integral_constant<T, v>) const {  
        typedef std::integral_constant<T, v> type;  
        (*this)(type::value);  
    }  
};
```

Simple variable length types

- Strings
- Arrays of integral types
- Etc.

Example string encoding

- Assume protocol represents strings as
 - uint16_t length
 - variable length char[]

Example string encoding

```
struct reader {  
    // ...  
  
    void operator()(std::string& val) const {  
        uint16_t length;  
        (*this)(length);  
        val = string(asio::buffer_cast<char const*>(buf_),  
                    length);  
        buf_ = buf_ + length;  
    }  
};
```

Example string encoding

```
struct writer {  
    // ...  
  
    void operator()(std::string const& val) const {  
        (*this)(static_cast<uint16_t>(val.length()));  
        asio::buffer_copy(buf_, asio::buffer(val));  
        buf_ = buf_ + val.length();  
    }  
};
```

Sequences of types

```
struct reader {  
    // ...  
  
    template<class T>  
    void operator()(std::vector<T> & vals) {  
        uint16_t length;  
        (*this)(length);  
        for (; length; --length) {  
            T val;  
            (*this)(val);  
            vals.emplace_back(std::move(val));  
        }  
    }  
};
```

Maps of types

```
#include <unordered_map>
```

```
struct reader {  
    // ...
```

```
    template<class K, class V>  
    void operator()(std::unordered_map<K,V> & kvs) {  
        uint16_t length;  
        (*this)(length);  
        for (; length; --length) {  
            K key;  
            (*this)(key);  
            V val;  
            (*this)(val);  
            kvs.emplace(key, val);  
        }
```

```
    }  
};
```


Multiple variable length fields

```
BOOST_FUSION_DEFINE_STRUCT(  
    (example), header,  
    (example::magic_t, magic)  
    (example::version_t, version)  
    (uint32_t, length)  
    (std::unordered_map<std::string, std::string>, hdr_props)  
    (example::msg_type_t, msg_type)  
    (std::vector<uint64_t>, vals)  
)
```

Framing

- Examples so far have ignored framing
 - For UDP you can continue to ignore it
 - TCP not so much

Framing

```
BOOST_FUSION_DEFINE_STRUCT(  
    (example), header,  
    (example::magic_t, magic)  
    (example::version_t, version)  
    (uint32_t, length)  
)
```

```
BOOST_FUSION_DEFINE_STRUCT(  
    (example), header_rest,  
    (std::map<std::string, std::string>, hdr_props)  
    (example::msg_type_t, msg_type)  
)
```

Framing

```
template<typename T>
T read(asio::const_buffer b) {
    reader r(std::move(b));
    T res;
    fusion::for_each(res, r);
    return res;
}
```

Framing

```
template<typename T>
std::pair<T, asio::const_buffer> read(asio::const_buffer b) {
    reader r(std::move(b));
    T res;
    fusion::for_each(res, r);
    return std::make_pair(res, r.buf_);
}
```

Framing

```
int main(int argc, char **argv) {  
    std::array<char, 1024> buf;  
    // receive header ...  
    example::header out;  
    asio::const_buffer buf_rest;  
    std::tie(out, buf_rest) = read<example::header>(asio::buffer(buf));  
  
    // receive out.length more bytes ...  
    example::header_rest vout;  
    std::tie(vout, buf_rest) = read<example::header_rest>(buf_rest);  
  
    // ...  
    return 0;  
}
```

User defined types

- Example decimal type
 - 8 bit signed exponent
 - 32 bit unsigned mantissa
- example $\{-2, 225\} = 2.25$
 - converted to a double

$$\textit{mantissa} * \textit{pow}(10, \textit{exponent})$$

User defined types

```
struct decimal_t {  
    int8_t exponent_;  
    uint32_t mantissa_;  
  
    decimal_t(int8_t e = 0, uint32_t m = 0)  
        : exponent_(e)  
        , mantissa_(m)  
    { }  
  
    operator double() const {  
        return mantissa_ * pow(10, exponent_);  
    }  
};
```


User defined types

```
struct reader {  
    // ...  
  
    void operator()(decimal_t & val) const {  
        int8_t e;  
        (*this)(e);  
        uint32_t m;  
        (*this)(m);  
        val = decimal_t(e, m);  
    }  
};
```

A non-trivial protocol

- Option definition
 - Option name e.g. GOOG14121567.5
 - Underlying stock e.g. GOOG
 - Strike price as decimal e.g. $\{-1, 5675\} = \$567.50$
 - Expiration date e.g. 2014-09-13 15:15:00
 - Put or Call - an enumeration
 - etc.

A non-trivial protocol

```
BOOST_FUSION_DEFINE_STRUCT(  
    (example), option_t,  
    (std::string, contract_id)  
    (std::string, underlying_id)  
    (example::decimal_t, strike)  
    (example::put_call_t, put_call)  
    (posix::date, expiration)  
)
```

A non-trivial protocol

- We also deal with things called “listed spreads”
 - The listed contract has an exchange assigned symbol/id
 - variable length collection of options contracts

A non-trivial protocol

```
BOOST_FUSION_DEFINE_STRUCT(  
    (example), spread_t,  
    (std::string, contract_id)  
    (std::vector<example::option_t>, legs)  
)
```

Nested Fusion structs

```
#include <boost/fusion/include/for_each.hpp>
#include <boost/fusion/include/is_sequence.hpp>

struct reader {
    // ...

    template<class T>
    auto operator()(T & val) const ->
    typename std::enable_if<is_sequence<T>::value>::type {
        boost::fusion::for_each(val, *this);
    }
};
```

Nested Fusion structs

```
#include <boost/fusion/include/for_each.hpp>
#include <boost/fusion/include/is_sequence.hpp>

struct writer {
    // ...

    template<class T>
    auto operator()(T const& val) const ->
    typename std::enable_if<is_sequence<T>::value>::type {
        boost::fusion::for_each(val, *this);
    }
};
```

Adapted Types

```
namespace example {  
    struct decimal_t {  
        int8_t exponent_;  
        uint32_t mantissa_;  
        // ...  
    };  
}  
  
BOOST_FUSION_ADAPT_STRUCT(  
    example::decimal_t,  
    (int8_t, exponent_)  
    (uint32_t, mantissa_)  
)
```


Fixing up read()

```
template<typename T>
std::pair<T, asio::const_buffer> read(asio::const_buffer b) {
    reader r(std::move(b));
    T res;
    fusion::for_each(res, r);
    return std::make_pair(res, r.buf_);
}
```

Fixing up read()

```
template<typename T>
std::pair<T, asio::const_buffer> read(asio::const_buffer b) {
    reader r(std::move(b));
    T res;
    r(res);
    return std::make_pair(res, r.buf_);
}
```

Optional fields

- Protocols may have optional fields
- Often indicated by bits in an integral type field

Optional Fields

```
namespace example {  
    template<typename T, size_t N = CHAR_BIT * sizeof(T)>  
    struct optional_field_set {  
        using value_type = T;  
        using bits_type = std::bitset<N>;  
    };  
  
    template<typename T, size_t N>  
    struct optional_field : boost::optional<T> {  
        constexpr static const size_t bit = N;  
    };  
  
    using opt_fields = optional_field_set<uint16_t>;  
    using opt_exercise = optional_field<exercise_t, 0>;  
    using opt_quote_ticks = optional_field<decimal_t, 1>;  
}
```

Optional Fields

```
BOOST_FUSION_DEFINE_STRUCT(  
    (example), option_t,  
    (std::string, contract_id)  
    (std::string, underlying_id)  
    (example::decimal_t, strike)  
    (example::put_call_t, put_call)  
    (posix::date, expiration)  
    (example::opt_fields, opts)  
    (example::opt_exercise, opt_a)  
    (example::opt_quote_ticks, opt_b)  
)
```

Optional Fields

```
struct reader {  
    mutable optional<opt_fields::bits_type> opts_  
  
    // ...  
  
    template<class T, size_t N>  
    void operator()(opt_fields) const {  
        opt_fields::value_type val;  
        (*this)(val);  
        opts_ = opt_fields::bits_type(val);  
    }  
};
```

Optional Fields

```
struct reader {  
    mutable optional<opt_fields::bits_type> opts_;  
  
    // ...  
  
    template<class T, size_t N>  
    void operator()(optional_field<T, N> & val) const {  
        if (!opts_)  
            throw bad_message();  
  
        if ((*opts_)[N]) {  
            T v;  
            (*this)(v);  
            val = v;  
        }  
    }  
};
```

Optional Fields

```
struct writer {  
    mutable asio::mutable_buffer buf_;  
    mutable example::opt_fields::bits_type opts_;  
    mutable example::opt_fields::value_type *optv_;  
  
    explicit writer(asio::mutable_buffer buf)  
        : buf_(std::move(buf))  
        , optv_(nullptr)  
    { }  
  
    // ...  
};
```


Optional Fields

```
struct writer {  
    // ...  
  
    void operator()(example::opt_fields) const {  
        opts_.reset();  
        optv_ = asio::buffer_cast<example::opt_fields::value_type*>(buf_);  
        buf_ = buf_ + sizeof(example::opt_fields::value_type);  
    }  
};
```

Optional Fields

```
struct writer {  
    // ...  
  
    template<class T, size_t N>  
    void operator()(example::optional_field<T, N> const& val) const {  
        if (!optv_)  
            throw bad_message();  
        if ((*opts_)[N]) {  
            opts_.set(N);  
            *optv_ = static_cast<opt_fields::value_type>(opts_.to_ulong());  
            (*this)(*val);  
        }  
    }  
};
```

Lazy/View Types

- Protocols may contain fields we may not always care about
- strings, maps, vectors all perform allocation, potentially expensive

String view

- Boost has a type called `string_ref`
 - Non-owning type with interface like `std::string`
 - implementation of `std::string_view`, likely to be in '17

std::string

```
struct reader {  
    // ...  
  
    void operator()(std::string& val) const {  
        uint16_t length = 0;  
        (*this)(length);  
        val = string(asio::buffer_cast<char const*>(buf_),  
                    length);  
        buf_ = buf_ + length;  
    }  
};
```

boost::string_ref

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

```
struct reader {  
    // ...
```

```
    void operator()(boost::string_ref & val) {
```

```
        uint16_t length = 0;
```

```
        (*this)(length)
```

```
        val = boost::string_ref(asio::buffer_cast<const char*>(buf_),  
                                length);
```

```
        buf_ = buf_ + length;
```

```
    }
```

```
};
```

Lazy types

- Need to be able to cheaply determine length
- Encode buffer position and length
- Decode on demand

A 'sizer'

```
struct sizer {
    mutable asio::const_buffer buf_;
    mutable size_t size_;

    explicit sizer(asio::const_buffer buf)
        : buf_(std::move(buf))
        , size_(0)
    { }

    template<class T>
    auto operator()(T &) const ->
    typename std::enable_if<std::is_integral<T>::value>::type {
        size_ += sizeof(T);
        buf_ = buf_ + sizeof(T);
    }

    // ...
};

template<class T>
size_t get_size(asio::const_buffer buf) { ... }
```


Generic lazy<T>

```
template<class T>
struct lazy {
    asio::const_buffer buf_;

    lazy(asio::const_buffer const& buf)
        : buf_(asio::buffer_cast<void const*>(buf),
              get_size<T>(buf))
    { }

    T get() const { return read<T>(buf_); }
    size_t size() const { return asio::buffer_size(buf_); }
};
```

Generic lazy<T>

```
struct reader {  
    // ...  
  
    template<class T>  
    void operator()(lazy<T> & val) const {  
        val = lazy<T>(buf_);  
        buf_ = buf_ + val.size();  
    }  
};
```

A non-trivial protocol

```
BOOST_FUSION_DEFINE_STRUCT(  
    (example), option_t,  
    (string_ref, contract_id)  
    (string_ref, underlying_id)  
    (example::decimal_t, strike)  
    (example::put_call_t, put_call)  
    (posix::date, expiration)  
)
```

```
BOOST_FUSION_DEFINE_STRUCT(  
    (example), spread_t,  
    (string_ref, contract_id)  
    (lazy<std::vector<example::option_t>>, legs)  
)
```

A non-trivial protocol

```
BOOST_FUSION_DEFINE_STRUCT(  
    (example), option_t,  
    (string_ref, contract_id)  
    (string_ref, underlying_id)  
    (example::decimal_t, strike)  
    (example::put_call_t, put_call)  
    (posix::date, expiration)  
)
```

```
BOOST_FUSION_DEFINE_STRUCT(  
    (example), spread_t,  
    (string_ref, contract_id)  
    (lazy_range<example::option_t>, legs)  
)
```

Getting field names

- Structure pretty-printing
- JSON, XML, etc.

Getting field names

```
#include <iostream>

int main(int argc, const char * argv[])
{
    example::spread_t s;
    // ...
    std::cout << example::to_json(s) << std::endl;
    return 0;
}
```

Getting field names

```
#include <boost/mpl/range_c.hpp>

template<class T>
struct printer {
    T & msg_;

    template<class T>
    using typename range_c = mpl::range_c<int, 0, mpl::size<T>::value>;

    friend std::ostream& operator<<(std::ostream & stm,
                                   printer<T> const& v) {
        mpl::for_each<range_c<T>>(mpl_visitor<T>(v.msg_, stm));
        return stm;
    }
};

template<class T>
printer<T> to_json(T const& v) { return printer<T>(v); }
```

Getting field names

```
#include <boost/mpl/for_each.hpp>
#include <boost/fusion/include/value_at.hpp>
#include <boost/fusion/include/at.hpp>

struct json_writer {
    // ...
};

template<class T>
struct mpl_visitor {
    T & msg_;
    std::ostream & stm_;
    json_writer writer_;

    mpl_visitor(std::ostream & stm_)
        : stm_(stm)
        , writer_(stm)
        { }

    // ...
};
```


Getting field names

```
namespace f_ext = boost::fusion::extension;
template<class T>
struct mpl_visitor {
    // ...

    template<class N>
    void operator()(N idx) {
        writer_(f_ext::struct_member_name<T, N::value>::call(), ":" );
        writer_(fusion::at<N>(msg_),
                (idx != mpl::size<T>::value ? "," : ""));
    }
};
```

Questions?

Sample code available at -

https://github.com/rodgert/fusion_samples/

Longer form version of this content -

<http://rodgert.github.io/2014/09/09/type-driven-wire-protocols-with-boost-fusion-pt1/>