Type Driven Wire Protocols with Boost Fusion

09 Sep 2014

Why not ProtoBufs, Thrift, etc.?

These solutions are great when you control both ends of the communication or when cross language compatibility is required. There are, however, many important use cases where this is not the case -

- 3rd party systems
- Embedded devices
- Legacy systems

So Why not Packed Structs?

The "classic C way" is to declare a packed struct which reflects the wire layout and then cast a pointer to a raw byte buffer to the desired type.

```
#include <array>
namespace example {
    struct header {
        uint16_t magic;
        uint16_t version;
        uint32_t length;
        uint32_t msg_type;
    } __attribute__((packed));
}

int main(int argc, char **argv) {
    std::array<char, sizeof(example::header)> buf;
    // ... read buf
    auto phdr = reinterpret_cast<example::header const*>(buf.data());
    // ... do something with phdr
}
```

This approach is simple and efficient; many fundamental protocol stacks are implemented this way (e.g. TCP, UDP, IP). It, however, has a few drawbacks -

- Depending on the hardware there may be substantial performance

penalties for member-wise access.
- You are limited to POD (plain old data-types).
- The types and layout of fields is restricted to the layout on the wire and likely does not match the desired types within the target problem domain.
- Explicit coding is required to fix up each member for byte ordering.
- This kind of code is often difficult to maintain and is generally not very modular or reusable.

In my day to day work, I deal with connectivity to financial markets which fall into the "3rd party systems" bucket. Frequently the type of data exchanged describes complex financial products with many nested variable length structures. Using the packed structure overlay approach quickly devolves into code that is hard to reason about, very difficult to maintain, and usually needs to be further wrapped into types that already exist in my application domain.

Is there a better way?

You could solve this by writing custom code generators, however in my experience the results tend to be fragile, consideration must be given to whether the intermediate artifacts are placed under source control, if not, how to integrate the code generator into the build, and the generator tends to be a hard to maintain one-off.

This being C++, we already have a wealth of inbuilt code-generating capability which might make it possible to sidestep some of the difficulties with creating a bespoke code generator.

It would be nice if Standard C++ had compile-time reflection

The primary capability we need is to enumerate the fields of a type, retrieving the field type and a reference to the field data. There is an active study group SG7 working on compile-time reflection proposals for eventual inclusion in the standard. Unfortunately Standard C++ does not yet provide such a facility.

What about Boost Fusion?

From the Boost Fusion Introduction -

> 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.

The important bits for our purposes are that Fusion allows us to work with both compile time containers of types and runtime containers of values.

A container of types and values -

```
#include <cstddef>

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

This *container* needs to be reworked a bit to get Fusion to generate the *magic boilerplate* necessary for our purposes -

```
#include <cstddef>
#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)
)
```

The BOOST_FUSION_DEFINE_STRUCT macro takes -

- a paren enclosed list of namespaces as (ns1)(ns2)...(nsx)
- a struct name
- a list of (type, field name) pairs

The networking code that I write makes extensive use of Boost Asio, and the sample code will make use of two types from Asio in particular -

- const_buffer - a safe non-modifiable buffer
- mutable_buffer - a safe modifiable buffer

Both types contain a **non-owning** pointer to underlying storage and a length. Each type exposes a '+' operator which returns a new buffer offset from the existing buffer by a given number of bytes. Buffers also support a number of free functions, the ones used here are -

- buffer_cast(buf) - returns a T* from the buffer
- buffer_size(buf) - returns the size of the buffer
- buffer_copy(dest, src) - copies bytes from one buffer to another
- buffer(T* v) - construct a buffer from pointer to T

Member-wise visitation

Now that we have a Fusion struct with *magic boilerplate*, we want to enumerate the contents and apply *some* operation to each member of the struct. Fusion uses the visitor pattern to achieve this -

```
#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) {
        // ...
    }
};

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_);
}
```

The reader type is the visitor, and the read() free function constructs a reader to read a given type from the supplied *const_buffer*. Unfortunately, this code is not quite right; fusion::for_each() takes the visitor by const reference, so we need to fix up the code. Specifically, we make the buf_ member *mutable* and the call operator() *const*, as follows -

```
#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 {
        // ...
    }
};

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_);
}
```

The code structure for writing data to a *mutable_buffer* is very similar -

```
#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(val, w);
    return w.buf_;
}
```

Fixing up byte ordering

Frequently the network byte order differs from the host machine byte order and must be fixed up on read and write. The standard functions for doing this are -

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

The *hton* variety convert unsigned 32 and 16 bit values from host to network byte order. The *ntoh* variety do the opposite conversion. There is a proposal to add a somewhat generic implementation for unsigned integral types to Standard C++. While an improvement, as currently proposed it is limited to unsigned types. What we really want is a generic implementation that works for all integral (signed or unsigned, >32 bit, etc.) types. It is fairly easy to roll our own (though I will skip the implementation details here), such that we can expect -

```
template<class T>
T hton(T v);

template<class T>
T ntoh(T v);
```

Work for the integral types we care about and make the appropriate byte order conversions for our wire protocol. We can amend our reader to now consume data from the *const_buffer* and set values in the supplied Fusion struct -

```
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);
    }
};
```

We can similarly amend the writer to write data to the *mutable_buffer* from values in the supplied Fusion struct -

```
struct writer {
    mutable asio::mutable_buffer buf_;
```

```
    // ...

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

Enumerated values

Frequently it is desirable to represent values as enums. C++11 added Scoped Enumerations which allow us to now specify the underlying type. This fits nicely into this technique. In our example header struct -

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

The struct member *msg_type* is an obvious candidate for declaring as an enumerated type -

```
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)
)
```

As it stands, our reader implementation is not quite sufficient. As currently defined -

```
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);
    }
};
```

The call operator() matches *any* type T, however ntoh/hton only work on *integral* types. We need to make the implementation more selective -

```
struct reader {
    // ...

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

We use the *std::enable_if<>* metafunction which allows us to conveniently leverage SFINAE to conditionally remove this overload when the type trait metafunction *std::is_integral<T>* does not evaluate to *std::true_type* for T.

We need to introduce a similar change to writer -

```
struct writer {
    // ...

    template<class T>
    auto operator()(T const& val) const ->
        typename std::enable_if<std::is_integral<T>::value>::type {
        T tmp = hton(val);
        asio::buffer_copy(buf_, asio::buffer(&tmp, sizeof(T)));
        buf_ = buf_ + sizeof(T);
    }
};
```

Now we can introduce a new overload for the call operator() to the reader which will handle enumeration types -

```
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);
    }
};
```

This overload uses *std::underlying_type<T>* to get the underlying type of the scoped enumeration then delegates the read to the matching overload for integral types. Finally, the result is cast to appropriate enum type.

We also need a similar change in the writer -

```
struct writer {
    // ...

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

Fixed tag data

Many protocols have fixed 'tag' data, this usually comes in the form of -

- *Magic* signature bytes
- Version markers

In our example header the aptly named *magic* and *version* fields are likely candidates for fixed tags.

```
#include <cstddef>
#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)
)
```

This data is constant there is no need to store it in our Fusion struct. All we need is a means of encoding type and expected value. C++11 introduced a type, *std::integral_constant<>* which does exactly this -

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

namespace example {
    using magic_t = std::integral_constant<uint16_t, 0xf00d>;
    using version_t = std::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)
)
```

We then add an overload to the reader -

```
struct reader {
    // ...

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

And a similar overload for the writer -

```
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

A drawback of the packed struct overlay approach is that you are generally left reading variable length content in multiple steps (e.g. read a header, read variable length type). The resulting struct definitions represent these sequences in terms of basic C types (e.g. char*, uint32_t*, etc,) rather than Standard C++ types such as strings, maps, vectors, or concepts such as ranges. We can do something a bit more *natural*.

Assuming string data is encoded as a 16 bit length prefix followed by some length number of chars, the reader is amended as follows -

```
#include <string>

struct reader {
   // ...

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

And for the writer -

```
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_ + length;
    }
};
```

Implementing support for *std::vector* is similarly straight forward -

```
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));
        }
    }
};

struct writer {
    // ...

    template<class T>
    void operator()(std::vector<T> const& vals) {
    (*this)(vals.length());
    for(auto&& val : vals)
        (*this)(val);
};
```

Implementing support for maps -

```
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);
        }
    }
};

struct writer {
    // ...

    template<class K, class V>
    void operator()(std::unordered_map<K, V> const& kvs) {
        (*this)(vals.length());
        for(auto& kv : kvs) {
            (*this)(kv.first);
            (*this)(kv.second);
        }
    }
};
```

It is possible to generically implement the writer for any type which conforms to

the ForwardRange concept.

```
struct writer {
    // ...

    template<class T, class U>
    void operator()(std::pair<T, U> const& val) {
        (*this)(val.first);
        (*this)(val.second);
    }

    template<typename T>
    auto operator()(T const& val) ->
        std::enable_if<boost::has_range_const_iterator<T>>::value>::t
ype {
        auto length = std::distance(std::begin(val), std::end(val));
        if (length > std::numeric_limits<uint16_t>::max())
            throw ...;

        (*this)(static_cast<uint16_t>(length));
        for (auto& val : vals)
            (*this)(val);
    }
};
```

Multiple variable length fields

Handling the following type with packed structs would be non-trivial -

```
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)
)
```

However, our reader/writer can already handle something like this.

Framing

All of the code so far has ignored the issue of framing. If the underlying transport is UDP (or some other message oriented transport which provides framing), we can continue to do so, however TCP is stream oriented and requires extra work. A typical approach for TCP wire protocols is to have a fixed length header which includes the total message length, so in practice, the above header would

typically be split as follows -

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

BOOST_FUSION_DEFINE_STRUCT(
    (example), header_rest,
    (std::unordered_map<std::string, std::string>, hdr_props)
)
```

A subsequent post will cover strategies for dealing with more complicated framing requirements.

User defined types

I work for a trading firm. One of the things we care about is the precise representation, in base-10, of prices. This means we need some other way besides float or double ($2.25 for instance is not precisely representable in base-2) of representing fixed point quantities. For the purposes of this sample we expect these values to be representable as -

- 8bit signed exponent
- 32bit unsigned mantissa

For example, { -2, 225 } is 2.25. The floating point approximation of the value is obtained by *mantissa * pow(10, exponent)*. We can represent this with the following UDT -

```
namespace example {
struct decimal_t {
    int8_t exponent_;
    uint32_t mantissa_;

    decimal_t(int8_t e, uint32_t m)
        : exponent_(e)
        , mantissa_(m)
    { }

    operator double() const { return mantissa_ * pow(10, exponent_);
}
};
} // namespace example
```

Extending our reader to handle such a type is straight forward -

```
struct reader {
    // ...

    void operator()(decimal_t * val) const {
        int8_t e;
        (*this)(e);
        uint32_t m;
        (*this)(m);
        val = decimal_t(e, m);
    }
};
```

This will work well enough for the moment, but we are going to revisit this design choice shortly.

Multiple nested variable length structures

The financial products I tend to work with most are options contracts. For our purposes, such a contract consists of -

- a contract identifier
- an identifier for the underlying product (e.g. stock)
- the strike price at which the contract is exerciseable
- expiration date of the contract
- an enumeration for the contract type, put or call

A representative Fusion struct would be -

```
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)
)
```

With the exception of a definition for handling *posix::date* fields (left as an exercise) we already have everything we need to support exchanging such types. However, we also deal with products called listed option spreads. For our purposes here, these consist of -

- a spread contract identifier

- a variable length list of options contracts that are part of the spread, aka the *legs*

A representative Fusion struct would be -

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

The only capability we are missing is ability to read and write nested Fusion structs. To do so we can amend the reader and writer as follows -

```
#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<boost::fusion::is_sequence<T>::value>::ty
pe {
        boost::fusion::for_each(val, *this);
    }
};

struct writer {
    // ...

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

With this change we can now deal with arbitrarily nested Fusion structs and we can revisit the approach for handling UDTs.

In addition to the BOOST_FUSION_DEFINE_STRUCT macro, there is a BOOST_FUSION_ADAPT_STRUCT macro which will create the *magic* boilerplate for existing types. We can change our handling of decimal_t as follows -

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

namespace example {
struct decimal_t {
    int8_t exponent_;
    uint32_t mantissa_;

    decimal_t(int8_t e, uint32_t m)
        : exponent_(e)
        , mantissa_(m)
    { }

    operator double() const { return mantissa_ * pow(10, exponent_);
}
};
} // namespace example

BOOST_FUSION_ADAPT_STRUCT(
    example::decimal_t,
    (int8_t, exponent_)
    (uint32_t, mantissa_)
)
```

We also need to remove the explicit overloads for decimal_t from the reader and writer and handle adapted UDTs with the overload for Fusion sequence types.

Optional Fields

Frequently protocols will have optional fields (often represented by a bit mask indicating which optional fields are present). Boost provides an *optional* type which can be either empty, or holds an instance of some type *T*. We can declare our optional fields generically using the following -

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

namespace example {
    template<typename T, size_t N>
    struct optional_field : boost::optional<T> {
        using boost::optional<T>::optional;
        constexpr static const size_t bit = N;
    };
} // namespace example
```

We also need some way to indicate to reader/writer where the field bitmask occurs. We can use a definition like this -

```
#include <bitset>
```

```
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>;
    };
} // namespace example
```

One thing to note about this type, it does not store any actual data. It simply acts as a *marker* or indication to the reader/writer of where the bitmask is to be expected.

With these definitions, we can now extend the reader as follows -

```
struct reader {
    mutable optional<opt_fields::bits_type> opts_;
    // ...

    template<class T, size_t N>
    void operator()(example::opt_fields) const {
        opt_fields::value_type val;
        (*this)(val);
        opts_ = opt_fields::bits_type(val);
    }

    template<class T, size_t N>
    void operator()(example::optional_field<T, N> & val) const {
        if (!opts_)
            throw ...
        if ((*opts_)[N]) {
            T v;
            (*this)(v);
            val = example::optional_field<T, N>(std::move(v));
        }
    }
};
```

Here, the optional reader member *opts_* is used to temporarily store the value of the bitmask when the reader encounters it. The overload for *optional_field* first verifies that opts_ has been set and then conditionally reads a value if the appropriate bit is set.

The writer case is a bit more interesting; we have to stash the location in the buffer where we write the bitmask when we encounter the opt_fields marker, and then update it with each optional field subsequently encountered.

```
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)
    { }

    // ...

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

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

With this capability in place we can now handle an option_t with optional fields -

```
namespace example {
    // ...
    enum class exercise_t : char {
        american = 'A', // default if not present
        european = 'E'
    };

    using opt_fields = optional_field_set<uint16_t>;
    using opt_exercise = optional_field<excercise_t, 0>;
    using opt_quote_ticks = optional_field<decimal_t, 1>;
} // namespace example

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)
)
```

Being Lazy

The examples so far have used types like *std::string*, *std::vector*, *std::map*, etc. These all require allocations (*std::map* is particularly awful in this regard) and copies. If we can guarantee the lifetime of the underlying Asio buffer exceeds the required lifetime for our structs, we can avoid or defer allocations and excess copying.

Strings

Boost contains an implementation of a non-owning *string_view* (Standard C++ proposal N3849 named string_ref. This type is largely a drop in replacement for *std::string* where we can guarantee the lifetime of the underlying buffer.

Our earlier implementation for std::string -

```
#include <string>

struct reader {
   // ...

   void operator()(std::string& val) const {
        uint16_t length;
        (*this)(length);
        val = std::string(asio::buffer_cast<char const*>(buf_), lengt
h);
        buf_ = buf + length;
    }
};
```

Becomes -

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

struct reader {
   // ...

   void operator()(boost::string_ref & val) const {
        uint16_t length;
        (*this)(length);
        val = boost::string_ref(asio::buffer_cast<char const*>(buf_),
length);
```

```
        buf_ = buf + length;
    }
};
```

We can then change the Fusion struct to use this type instead -

```
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)
)
```

Lazily constructed types

We might want to defer the cost of construction until we actually need the contents of a particular field. With same lifetime guarantees we assume in the case *string_ref*, it is possible to build a type which will do this -

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

//...

void process_legs(example::spread_t const& s) {
    for(auto&& leg : s.legs.get()) {
        // ...
    }
}
```

Our *lazy<T>* needs to -

- cheaply determine the length of *T*
- encode buffer position and length
- decode and construct on demand

Computing the size of T in a buffer is *almost* the same as reading T from the buffer -

```
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) {
    sizer s(std::move(buf));
    T val;
    s(val);
    return s.size_;
}
```

With the sizer in place, the *lazy<T>* implementation is straight forward -

```
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))
    {
        buf = buf + asio::buffer_size(buf_);
    }

    T get() const { return read<T>(buf_); }

    size_t size() const { return asio::buffer_size(buf_); }
};
```

The last step is adding an overload to the reader -

```
struct reader {
    // ...

    template<class T>
    void operator()(lazy<T> & val) const {
```

```
        val = lazy<T>(buf_);
        buf_ = buf_ + val.size();
    }
};
```

Getting field names and structure pretty-printing

All examples so far have demonstrated getting the type and reference to a field within a Fusion struct. There are times when we would like to get access to field names as a string. Fusion provides a way to get at this, but it is not particularly well documented and the approach deviates a bit from what we have done so far. Fusion has an extension function template called *struct_member_name<T, N>* where the two template arguments are -

- T - the Fusion struct
- N - the index of the field within the Fusion struct for which we wish to retrieve the name

The Fusion visitor function *for_each()* unfortunately does not give us a usable N, which must be a constant known at compile time, to supply to *struct_member_name<>*. We can however leverage the fact that every Fusion sequence is an MPL sequence to get us where we need to be -

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

namespace detail {
    namespace mpl = boost::mpl;
    template<class T>
    using typename range_c = mpl::range_c<int, 0, mpl::size<T>::value
>;
};
```

Here the range_c type will generate a *compile-time* sequence of integral values from 0 to the length of the supplied boost::mpl sequence. This definition is used to construct a printer type that exposes an stream insertion operator<<() which will visit each field index of a given Fusion type -

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

#include <ostream>

namespace detail {
    // ...
    template<class T>
    struct mpl_visitor {
```

```
        // ...
    };
}

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

    // ...

    friend std::ostream& operator<<(std::ostream& stm, printer<T> con
st& v) {
        using namespace detail;
        boost::mpl::for_each<typename range_c<T>>(mpl_visitor<T>(stm,
v.msg_));
        return stm;
    }
};
```

Now define the mpl_visitor detail -

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

namespace detail {
    struct value_writer {
        // ...
    };

    // ...

    namespace f_ext = boost::fusion::extension;
    template<class T>
    struct mpl_visitor {
        value_writer w_;
        T & msg_;

        mpl_visitor(std::ostream & stm, T & msg)
            : value_writer_(stm)
            , msg_(msg)
        { }

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

Lastly, define a free function to return an instance of *printer<T>*

```
// ...

template<class T>
detail::printer<T> pretty_print(T const& v) {
    return detail::printer<T>(v);
}

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

Related Posts