

Make Classes Great Again!

Vinnie Falco
Author of Boost.Beast

CppCon 2017
Sep 25-29



Boost.Beast

- HTTP and WebSocket protocols
- Using Boost.Asio
- Header-only C++11
- Coming to Boost 1.66.0

<https://github.com/boostorg/beast>

Boost C++ Libraries

- Establish existing practice
- Become part of C++

`boost::shared_ptr`

`boost::optional`

`boost::bind`

`boost::mutex`

`boost::chrono`

`BOOST_FOREACH`

`boost::asio`

`boost::filesystem`

`boost::thread`

`boost::shared_mutex`

`boost::function`

`BOOST_STATIC_ASSERT`



Outline

1. HTTP Primer
2. Message Model
3. Define Concept
4. Documentation
5. Type Traits

HTTP

- “Hypertext Transfer Protocol”

Clients



`$curl http://example.com`



Servers

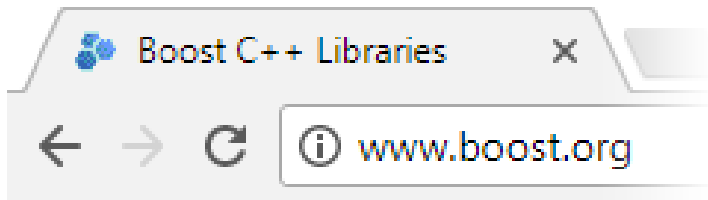


NGINX



HTTP

Client



DNS



Server

192.168.17.42



TCP/IP



HTTP

Client

Server

HTTP Request



HTTP Response

HTTP Request

```
GET /index.html HTTP/1.1  
User-Agent: Chrome  
Host: www.boost.org  
Accept-Language: en-us  
Accept-Encoding: gzip  
Connection: Keep-Alive
```

HTTP Request

VERB	TARGET	VERSION	REQUEST LINE
GET	/index.html	HTTP/1.1	
User-Agent: Chrome			
Host: www.boost.org			
Accept-Language: en-us			
Accept-Encoding: gzip			
Connection: Keep-Alive			

HTTP Request

GET /index.html HTTP/1.1

User-Agent: Chrome

Host: www.boost.org

Accept-Language: en-us

Accept-Encoding: gzip

Connection: Keep-Alive

NAME

VALUE

FIELDS

Field names are case-insensitive

HTTP Response

HTTP/1.1 200 OK

Date: Sun, 18 Oct 2012

Server: Apache/2.2.14

Connection: close

Content-Type: text/html

Content-Length: 55

<html><head></head><body>welcome
to Boost!</body></html>

HTTP Response

VERSION CODE REASON STATUS LINE
HTTP/1.1 200 OK

Date: Sun, 18 Oct 2012

Server: Apache/2.2.14

Connection: close

Content-Type: text/html

Content-Length: 55

<html><head></head><body>Welcome
to Boost!</body></html>

HTTP Response

HTTP/1.1 200 OK

Date: Sun, 18 Oct 2012

Server: Apache/2.2.14

Connection: close

Content-Type: text/html

Content-Length: 55

FIELDS

<html><head></head><body>welcome
to Boost!</body></html>

HTTP Response

```
HTTP/1.1 200 OK  
Date: Sun, 18 Oct 2012  
Server: Apache/2.2.14  
Connection: close  
Content-Type: text/html  
Content-Length: 55
```

```
<html><head></head><body>welcome  
to Boost!</body></html>
```

BODY

HTTP Message

Start-Line: Request-Line or Status-Line

Fields: zero or name/value pairs

Body: optional

HTTP Message

HEADER

Start-Line: Request-Line or Status-Line

Fields: zero or name/value pairs

Body: optional

HTTP Message

Start-Line: Request-Line or Status-Line

Fields: zero or name/value pairs

Body: optional

BODY

HTTP Message

- Model a message in C++
- “Get” and “Set” attributes
- Serialize (to network format)
- Parse (from network format)

Message Model

```
/// Holds an HTTP request
struct request
{
    int            version; // 10 or 11
    string         method;  // e.g. "GET", "POST"
    string         target;  // "/index.html"
    map<string,string> fields; // name/value pairs
    string         body;    // variable size
};
```

(Data members are used instead of member functions, for brevity)

Message Model

```
/// Holds an HTTP response
struct response
{
    int          version; // 10 or 11
    int          status;  // 200 means OK
    string       reason;  // human readable
    map<string, string> fields; // name/value pairs
    string       body;    // variable size
};
```

(Data members are used instead of member functions, for brevity)

Message Model

```
/// Holds an HTTP response
struct response
{
    int          version; // 10 or 11
    int          status;  // 200 means OK
    string       reason;  // human readable
    map<string,string> fields; // name/value pairs
    string       body;    // variable size
};
```



Message Model

- Not AllocatorAware
- body is fixed to `std::string`
- fields is fixed to `std::map`
- Field names are case-insensitive
- request, response types distinct/unrelated:

```
/// Serialize an HTTP message
template<class Message>
void write(ostream&, Message const&);
```


Message Class

```
/// Holds an HTTP message
template<bool isRequest>
struct message;                // class template

/// Holds an HTTP request
template<>
struct message<true>           // specialization
{
    int                version;
    string              method;
    string              target;
    map<string,string>  fields;
    string              body;
};
```

Message Class

```
/// Holds an HTTP message
template<bool isRequest>
struct message;                // class template

/// Holds an HTTP request
template<>
struct message<true>           // specialization
{
    int                version;
    string              method;
    string              target;
    map<string, string> fields;
    string              body;
};
```

Message Class

```
/// Holds an HTTP request  
using request = message<true>;
```

```
/// Holds an HTTP response  
using response = message<false>;
```

```
/// Serialize an HTTP message  
template<bool isRequest>  
void write(ostream&  
           message<isRequest> const&);
```

Message Class

```
/// Holds an HTTP request  
using request = message<true>;
```

```
/// Holds an HTTP response  
using response = message<false>;
```

```
/// Serialize an HTTP message  
template<bool isRequest>  
void write(ostream&  
           message<isRequest> const&);
```

AllocatorAware

- Not AllocatorAware

/// Holds an HTTP request

```
template<>
```

```
struct message<true>
```

```
{
```

```
    int
```

```
    string
```

```
    string
```

```
    map<string, string>
```

```
    string
```

```
};
```

```
    version;
```

```
    method;
```

```
    target;
```

```
    fields;
```

```
    body;
```



AllocatorAware

```
/// Holds an HTTP request
template<class OuterAlloc, class InnerAlloc = OuterAlloc,
        class StringAlloc = OuterAlloc, class BodyAlloc = OuterAlloc>
struct message<true, OuterAlloc, InnerAlloc, StringAlloc, BodyAlloc>
{
    int version;
    basic_string<char, char_traits<char>, StringAlloc> method;
    basic_string<char, char_traits<char>, StringAlloc> target;

    using inner_string = basic_string<
        char, char_traits<char>, InnerAlloc>;
    map<inner_string, inner_string, less<inner_string>,
        scoped_allocator_adaptor<OuterAlloc, InnerAlloc>> fields;

    basic_string<char, char_traits<char>, BodyAlloc> body;
};
```

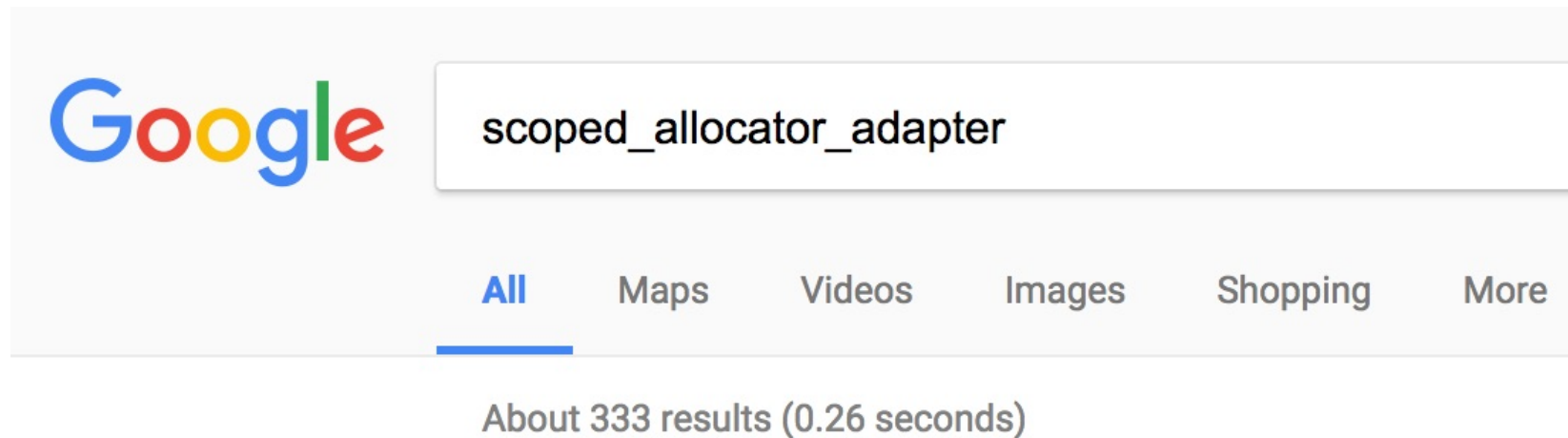
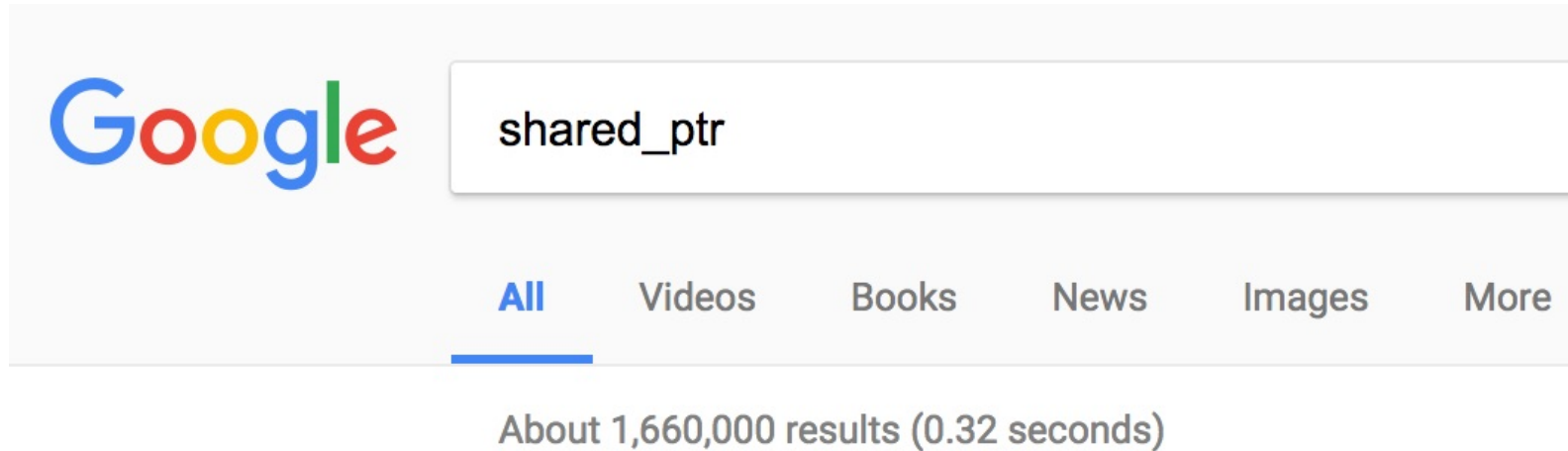
AllocatorAware

```
/// Holds an HTTP request
template<class OuterAlloc, class InnerAlloc = OuterAlloc,
        class StringAlloc = OuterAlloc, class BodyAlloc = OuterAlloc>
struct message<true, OuterAlloc, InnerAlloc, StringAlloc, BodyAlloc>
{
    int version;
    basic_string<char, char_traits<char>, StringAlloc> method;
    basic_string<char, char_traits<char>, StringAlloc> target;

    using inner_string = basic_string<
        char, char_traits<char>, InnerAlloc>;
    map<inner_string, inner_string, less<inner_string>,
        scoped_allocator_adaptor<OuterAlloc, InnerAlloc>> fields;

    basic_string<char, char_traits<char>, BodyAlloc> body;
};
```

AllocatorAware



AllocatorAware

- Allocators are hard, skip it for now
- Customize the body instead

Body Customization

“All problems in computer science can be solved by another level of indirection”

- David Wheeler

Body Customization

```
/// Holds an HTTP request
template<>
struct message<true>
{
    int                version;
    string             method;
    string             target;
    map<string,string> fields;

    string             body;
};
```



Body Customization

```
/// Holds an HTTP request
template<class Body>
struct message<true>
{
    int                version;
    string             method;
    string             target;
    map<string,string> fields;

    Body               body;
};
```



Body Customization

```
/// Holds an HTTP message
template<bool isRequest, class Body>
struct message;

/// Holds an HTTP request
template<class Body>
using request = message<true, Body>;

/// Holds an HTTP response
template<class Body>
using response = message<false, Body>;
```


Body Customization

```
/// Holds an HTTP message  
template<bool isRequest, class Body>  
struct message;
```

```
/// Holds an HTTP request  
template<class Body>  
using request = message<true, Body>;
```

```
/// Holds an HTTP response  
template<class Body>  
using response = message<false, Body>;
```

Body Customization

```
// A message that uses a string body
request<string> req;
req.body = "Hello, world!";
```

```
// A message that uses a vector body
response<vector<char>> res;
res.body = { 'a', 'b', 'c' };
```

```
/// Serialize an HTTP message
template<bool isRequest, class Body>
void write(ostream&,
           message<isRequest, Body> const& msg);
```

Body Customization

```
/// Serialize an HTTP message
template<bool isRequest, class Body>
void write(ostream& os, message<isRequest, Body> const& msg)
{
    write_header(os, msg);
    os.write(msg.body.data(), msg.body.size());
}
```

```
request<string> req;
write(cout, req);
```

```
response<vector<char>> res;
write(cout, res);
```

```
response<list<string>> res;
write(cout, res);
```

Body Customization

```
/// Serialize an HTTP message
template<bool isRequest, class Body>
void write(ostream& os, message<isRequest, Body> const& msg)
{
    write_header(os, msg);
    os.write(msg.body.data(), msg.body.size());
}
```

```
request<string> req;
write(cout, req);
```

```
response<vector<char>> res;
write(cout, res);
```

```
response<list<string>> res;
write(cout, res);
```

Body Customization

```
/// Serialize an HTTP message
template<bool isRequest, class Body>
void write(ostream& os, message<isRequest, Body> const& msg)
{
    write_header(os, msg);
    os.write(msg.body.data(), msg.body.size());
}
```

```
request<string> req;
write(cout, req);
```

```
response<vector<char>> res;
write(cout, res);
```

```
response<list<string>> res;
write(cout, res);
```

Body Customization

```
/// Serialize an HTTP message
template<bool isRequest, class Body>
void write(ostream& os, message<isRequest, Body> const& msg)
{
    write_header(os, msg);
    os.write(msg.body.data(), msg.body.size());
}
```

```
request<string> req;
write(cout, req);
```

```
response<vector<char>> res;
write(cout, res);
```

```
response<list<string>> res;
write(cout, res);
```

Body Customization

```
/// Serialize an HTTP message
template<bool isRequest, class Body>
void write(ostream& os, message<isRequest, Body> const& msg)
{
    write_header(os, msg);
    os.write(msg.body.data(), msg.body.size());
}
```

```
request<string> req;
write(cout, req);
```

```
response<vector<char>> res;
write(cout, res);
```

```
response<list<string>> res;
write(cout, res);
```

Body Customization

```
/// Serialize an HTTP message
template<bool isRequest, class Body>
void write(ostream& os, message<isRequest, Body> const& msg)
{
    write_header(os, msg);
    os.write(msg.body.data(), msg.body.size());
}

request<string> req;
write(cout, req);

response<vector<char>> res;
write(cout, res);

response<list<string>> res;
write(cout, res);
```


Body Customization

```
/// Serialize an HTTP message
template<bool isRequest, class Body>
void write(ostream& os, message<isRequest, Body> const& msg)
{
    write_header(os, msg);
    os.write(msg.body.data(), msg.body.size());
}
```

```
request<string> req;
write(cout, req);
```

```
response<vector<char>> res;
write(cout, res);
```

```
response<list<string>> res;
write(cout, res); // WAT?
```



Body Customization

- Oops! no `std::list::data()`

```
// write() can't work with this type!
```

```
request<list<string>> req;
```

```
// Works: message body is already in memory
```

```
request<string> req;
```

```
req.body = "<html><head></head><body>Hello!</body></head>";
```

```
write(cout, req);
```

```
// Send a file in the response
```

```
response<string> res;
```

```
res.body = "C:\\Users\\Vinnie\\www\\index.html";
```

```
// Doesn't do the right thing!
```

```
write(cout, res);
```

Body Customization

- Oops! no `std::list::data()`

```
// write() can't work with this type!
```

```
request<list<string>> req;
```

```
// Works: message body is already in memory
```

```
request<string> req;
```

```
req.body = "<html><head></head><body>Hello!</body></head>";
```

```
write(cout, req);
```

```
// Send a file in the response
```

```
response<string> res;
```

```
res.body = "C:\\Users\\Vinnie\\www\\index.html";
```

```
// Doesn't do the right thing!
```

```
write(cout, res);
```

Body Customization

- Oops! no `std::list::data()`

```
// write() can't work with this type!
```

```
request<list<string>> req;
```

```
// Works: message body is already in memory
```

```
request<string> req;
```

```
req.body = "<html><head></head><body>Hello!</body></head>";
```

```
write(cout, req);
```

```
// Send a file in the response
```

```
response<string> res;
```

```
res.body = "C:\\Users\\Vinnie\\www\\index.html";
```

```
// Doesn't do the right thing!
```

```
write(cout, res);
```

Body Customization

- Oops! no `std::list::data()`

```
// write() can't work with this type!
```

```
request<list<string>> req;
```

```
// Works: message body is already in memory
```

```
request<string> req;
```

```
req.body = "<html><head></head><body>Hello!</body></head>";
```

```
write(cout, req);
```

```
// Send a file in the response
```

```
response<string> res;
```

```
res.body = "C:\\Users\\Vinnie\\www\\index.html";
```

```
// Doesn't do the right thing!
```

```
write(cout, res);
```

Body Customization

```
/// Request with a text body
request<string> req1;
req1.body = "Hello, world!";
```

```
/// Request with an empty body
request<string> req2;
req2.body.clear();
```

```
assert(sizeof(req2) < sizeof(req1)); // Fails
```

```
/// Serialize an HTTP message
template<bool isRequest, class Body>
void write(ostream& os, message<isRequest, Body> const& msg)
{
    write_header(os, msg);
    os.write(msg.body.data(), msg.body.size()); // Not zero cost
}
```

Body Customization

```
/// Request with a text body
request<string> req1;
req1.body = "Hello, world!";
```

```
/// Request with an empty body
request<string> req2;
req2.body.clear();
```

```
assert(sizeof(req2) < sizeof(req1)); // Fails
```

```
/// Serialize an HTTP message
template<bool isRequest, class Body>
void write(ostream& os, message<isRequest, Body> const& msg)
{
    write_header(os, msg);
    os.write(msg.body.data(), msg.body.size()); // Not zero cost
}
```

Body Customization

“All problems in computer science can be solved by another level of indirection”

- David Wheeler

Body Type

```
/// Holds an HTTP request
template<class Body>
struct message<true, Body>
{
    int                version;
    string             method;
    string             target;
    map<string, string> fields;

    Body              body;
};
```



Body Type

```
/// Holds an HTTP request
template<class Body>
struct message<true, Body>
{
    int                version;
    string             method;
    string             target;
    map<string, string> fields;

    typename Body::value_type body;
};
```



Body Type

```
/// A Body that uses a string container
struct string_body
{
    using value_type = string;
};

// Use a string for the body container
response<string_body> res;
res.body = "Hello, world!";
```

Body Type

/// A Body that uses a vector container

```
template<class T>
struct vector_body
{
    using value_type = vector<T>;
};
```

// Use a vector for the body container

```
response<vector_body<char>> res;
res.body = { 'a', 'b', 'c' };
```

Body Type

```
/// A Body that uses a list container
```

```
template<class T>  
struct list_body  
{  
    using value_type = list<T>;  
};
```

```
// Use a list for the body container
```

```
response<list_body<string>> res;  
res.body = { "Hello ", "world!" };
```

```
// Still doesn't work  
response<list<string>> res;  
write(cout, res);
```

Body Type

```
/// A Body that uses a list container
```

```
template<class T>  
struct list_body  
{  
    using value_type = list<T>;  
};
```

```
// Use a list for the body container
```

```
response<list_body<string>> res;  
res.body = { "Hello ", "world!" };
```

```
// Still doesn't work
```

```
response<list<string>> res;  
write(cout, res);
```

Body Type

“All problems in computer science can be solved by another level of indirection”

- David Wheeler

Body Type

```
/// Serialize an HTTP message
template<bool isRequest, class Body>
void write(ostream& os,
           message<isRequest, Body> const& msg)
{
    write_header(os, msg);
    os.write(msg.body.data(), msg.body.size());
}
```



Body Type

```
/// Serialize an HTTP message
template<bool isRequest, class Body>
void write(ostream& os,
           message<isRequest, Body> const& msg)
{
    write_header(os, msg);
    Body::write(os, msg.body);
}
```



Body Type

/// A Body that uses a string container

```
struct string_body
{
    using value_type = string;

    static void write(
        ostream& os, string const& body)
    {
        os << body;
    }
};
```

// Use a string for the body container

```
response<string_body> res;
res.body = "Hello, world!";
```

Body Type

```
/// A Body that uses a list container
template<class T>
struct list_body
{
    using value_type = list<T>;

    static void write(
        ostream& os, list<T> const& body)
    {
        for(auto const& value : body)
            os << value;
    }
};

// Use a list of strings for the body container
response<list_body<string>> res;
res.body = { "Hello ", "world!" };
```

Body Type

```
/// An empty Body  
struct empty_body;
```

Body Type

```
/// An empty Body
struct empty_body
{
    struct value_type {};

    static void write(ostream&, value_type const&)
    {
        // Do nothing
    }
};
```

```
// A request with no body
request<empty_body> req;
assert(sizeof(req.body) == 0); // fails
```

Body Type

```
/// An empty Body
struct empty_body
{
    struct value_type {};

    static void write(ostream&, value_type const&)
    {
        // Do nothing
    }
};
```

```
// A request with no body
request<empty_body> req;
assert(sizeof(req.body) == 0); // fails
```



Body Type

```
/// Holds an HTTP request
template<class Body>
struct message<true, Body>
{
    int                version;
    string             method;
    string             target;
    map<string,string> fields;

    typename Body::value_type& body();
    typename Body::value_type const& body() const;
};
```

Body Type

```
/// Holds an HTTP request
template<class Body>
struct message<true, Body> :
    private Body::value_type // empty base optimization
{
    int                version;
    string             method;
    string             target;
    map<string,string> fields;

    typename Body::value_type& body()
    {
        return *this;
    }

    typename Body::value_type const& body() const
    {
        return *this;
    }
};
```


Body Type

```
// Send a file in the response  
response<file_body> res;  
res.body = "C:\\Users\\Vinnie\\www\\index.html";  
write(cout, res);
```



Body Type

```
/// A Body using file contents
struct file_body
{
    using value_type = string; // Path to file

    static void write(
        ostream& os, string const& path)
    {
        size_t n;
        char buf[4096];
        FILE* f = fopen(path.c_str(), "rb");
        while(n = fread(buf, 1, sizeof(buf), f))
            os.write(buf, n);
        fclose(f);
    }
};
```

Concept

- For a given template type, a Concept defines:
 - Syntax requirements
(Correct compilation)
 - Semantic requirements
(Correct behavior)



Concept

// Types implementing the Body concept:

```
struct string_body;
```

```
template<class T>  
struct vector_body;
```

```
template<class T>  
struct list_body;
```

```
struct file_body;
```

```
struct empty_body;
```

Concept

- `#include <algorithm>`

`std::swap`

`std::lock`

`std::find`

`std::sort`

`std::max`

Concept

- `#include <algorithm>`

`std::swap`

Swappable

`std::lock`

Lockable

`std::find`

InputIterator

`std::sort`

RandomAccessIterator

`std::max`

LessThanComparable

Documentation

```
/// Holds an HTTP message
template<
    bool isRequest,
    class Body>      // What type goes here?
struct message;
```

```
// What type is used for Body?
template<class Body>
using request = message<true, Body>;
```

Documentation

```
// Exemplar:
//
//      A Body defines the type of container
//      and algorithm used to represent the
//      body in an HTTP message.
struct Body
{
    // The type of the message::body member
    class value_type;

    // The algorithm for serializing this body type
    static void write(
        ostream& os, value_type const& body);
};
```


Documentation

A **Body** defines the type of container and algorithm used to represent the body in an HTTP message.

In this table:

B is a type meeting the requirements of **Body**.

Requirements:

Expression	Type	Description
<code>B::value_type</code>		The type of container used to represent the body in a message.
<code>B::write</code>	<code>void(ostream& B::value_type const&)</code>	A function invoked by the implementation to serialize the body to a <code>std::ostream</code> .

Metafunctions

Disclaimer

The following examples show what is possible using simple C++11 and are meant for exposition only. They do not represent the latest C++17 and later features; specifically, the “detection idiom toolkit:”

```
#include <experimental/type_traits>
```

```
template<template<class...> class Op, class... Args>  
using is_detected = ... ;
```

```
template<template<class...> class Op, class... Args>  
using detected_t = ... ;
```

```
template<class Default, template<class...> class Op, class... Args>  
using detected_or = ... ;
```

Metafunctions

- Determine if B meets the requirements of **Body**

```
/// Alias for true_type if B is a Body
```

```
template<class B>  
struct is_body;
```

```
/// Serialize an HTTP message  
template<bool isRequest, class Body>  
void write(ostream& os,  
           message<isRequest, Body> const& msg)  
{  
    static_assert(is_body<Body>::value,  
                  "Body requirements not met");  
    write_header(os, msg);  
    Body::write(os, msg.body);  
}
```

Metafunctions

- Determine if B meets the requirements of **Body**

```
/// Alias for true_type if B is a Body
```

```
template<class B>  
struct is_body;
```

```
/// Serialize an HTTP message
```

```
template<bool isRequest, class Body>  
void write(ostream& os,  
           message<isRequest, Body> const& msg)  
{  
    static_assert(is_body<Body>::value,  
                  "Body requirements not met");  
    write_header(os, msg);  
    Body::write(os, msg.body);  
}
```

Metafunctions

```
// Maps types to void
template<class...>
using void_t = void; // (since C++17) modulo compiler bugs

// Primary template catches any type
// without a nested ::value_type member
template<class B, class = void>
struct is_body : false_type {};

// Catches B with a nested ::value_type
template<class B>
struct is_body<B, void_t<
    typename B::value_type
    >> : true_type {};
```

Metafunctions

```
// Maps types to void
template<class...>
using void_t = void; // (since C++17) modulo compiler bugs
```

```
// Primary template catches any type
// without a nested ::value_type member
template<class B, class = void>
struct is_body : false_type {};
```

```
// Catches B with a nested ::value_type
template<class B>
struct is_body<B, void_t<
    typename B::value_type
    >> : true_type {};
```

Metafunctions

```
// Maps types to void
template<class...>
using void_t = void; // (since C++17) modulo compiler bugs
```

```
// Primary template catches any type
// without a nested ::value_type member
template<class B, class = void>
struct is_body : false_type {};
```

```
// Catches B with a nested ::value_type
template<class B>
struct is_body<B, void_t<
    typename B::value_type           // SFINAE applies here
    >> : true_type {};
```

Metafunctions

```
// Require a nested ::value_type, and a static
// write() member with the correct signature
template<class B>
struct is_body<B, void_t<
    typename B::value_type,
    decltype(
        B::write(
            declval<ostream&>(),
            declval<typename B::value_type const&>()),
            (void)0)
    >> : true_type {};
```


Generic Programming

- Template functions, template classes
- Template parameter types are Concepts
- Concepts have documentation:
 - Syntactic requirements
 - Semantic requirements:
 - pre-conditions, post-conditions
 - exception guarantees
 - algorithmic complexity
 - Compile-time introspection (e.g. traits)

Generic Programming

```
/// Serialize an HTTP message
template<bool isRequest, class Body>
void write(
    ostream& os,
    message<isRequest, Body> const& msg)
```

Parsing

```
/// Parse an HTTP message  
template<bool isRequest, class Body>  
void read(istream& is,  
          message<isRequest, Body>& msg)
```



Parsing

```
/// Parse an HTTP message
template<bool isRequest, class Body>
void read(istream& is,
          message<isRequest, Body>& msg)
{
    read_header(is, msg);
    Body::read(is, msg.body);
}
```

Parsing

/// A Body that uses a string container

```
struct string_body
{
    using value_type = string;

    static void read(
        istream& is, string& body)
    {
        is >> body;
    }

    static void write(
        ostream& os, string const& body)
    {
        os << body;
    }
};
```

Parsing

A **Body** defines the type of container and algorithm used to represent the body in an HTTP message.

In this table:

B is a type meeting the requirements of **Body**.

Requirements:

Expression	Type	Description
<code>B::value_type</code>		The type of container used to represent the body in a message.
<code>B::read</code>	<code>void(istream& B::value_type&)</code>	A function invoked by the implementation to parse the body from a <code>std::istream</code> .
<code>B::write</code>	<code>void(ostream& B::value_type const&)</code>	A function invoked by the implementation to serialize the body to a <code>std::ostream</code> .

Parsing

```
// Require a nested ::value_type, and static read()
// and write() members with the correct signature
template<class B>
struct is_body<B, void_t<
    typename B::value_type,
    decltype(
        B::read(
            declval<istream&>(),
            declval<typename B::value_type&>()),
        B::write(
            declval<ostream&>(),
            declval<typename B::value_type const&>()),
        (void)0)
>> : true_type {};
```

AllocatorAware

// Use a vector for the body container

```
response<vector_body<char>> res;
```

/// A Body that uses a vector container

```
template<class T, class Allocator = allocator<T>>
```

```
struct vector_body
```

```
{
```

```
    using value_type = vector<T, Allocator>;
```

```
    static void write(
```

```
        ostream& os, value_type const& body);
```

```
};
```


AllocatorAware

```
// Use a vector for the body container
```

```
response<vector_body<char>> res;
```

```
/// A Body that uses a vector container
```

```
template<class T, class Allocator = allocator<T>>
```

```
struct vector_body
```

```
{
```

```
    using value_type = vector<T, Allocator>;
```

```
    static void write(
```

```
        ostream& os, vector<T, Allocator> const& body);
```

```
};
```

AllocatorAware

```
// Define an instance of our custom allocator  
my_alloc a{65536};
```

```
// A response that uses a custom allocator  
response<vector_body<char, my_alloc>> res{a}; // ?
```



AllocatorAware

```
/// Holds an HTTP response
template<class Body>
struct message<false, Body>
{
    int                version;
    int                status;
    string             reason;
    map<string, string> fields;
    typename Body::value_type body;

    /// Arguments forwarded to body constructor
    template<class... Args>
    explicit message(Args&&... args)
        : body(forward<Args>(args)...)
    {}
};
```

Fields

```
/// Holds an HTTP response
template<class Body>
struct message<false, Body>
{
    int          version;
    int          status;
    string       reason;
    map<string, string> fields;

    typename Body::value_type body;
};
```



Fields

```
/// Holds an HTTP response
template<class Body, class Fields>
struct message<false, Body> : Fields
{
    int                version;
    int                status;
    string             reason;

    typename Body::value_type body;
};
```



Fields

```
/// Holds an HTTP message
```

```
template<  
    bool isRequest,  
    class Body,  
    class Fields>  
struct message;
```

```
/// Serialize an HTTP message
```

```
template<bool isRequest, class Body, class Fields>  
void write(  
    ostream& os,  
    message<isRequest, Body, Fields> const& msg)
```

Fields

/// Holds an HTTP message

```
template<
    bool isRequest,
    class Body,
    class Fields> // What type goes here?
struct message;
```

/// Serialize an HTTP message

```
template<bool isRequest, class Body, class Fields>
void write(
    ostream& os,
    message<isRequest, Body, Fields> const& msg)
```



Fields

```
/// An associative container for HTTP fields
template<class Allocator>
struct basic_fields
{
    void set(string_view name, string_view value);
    string_view operator[](string_view name) const;
};
```

```
/// A typical HTTP Fields container
using fields = basic_fields<allocator<char>>;
```

```
/// Holds an HTTP request
template<class Body, class Fields = fields>
using request = message<true, Body, Fields>;
```

```
/// Holds an HTTP response
template<class Body, class Fields = fields>
using response = message<false, Body, Fields>;
```


Fields

```
/// An associative container for HTTP fields
template<class Allocator>
struct basic_fields
{
    void set(string_view name, string_view value);
    string_view operator[](string_view name) const;
};
```

```
/// A typical HTTP Fields container
using fields = basic_fields<allocator<char>>;
```

```
/// Holds an HTTP request
template<class Body, class Fields = fields>
using request = message<true, Body, Fields>;
```

```
/// Holds an HTTP response
template<class Body, class Fields = fields>
using response = message<false, Body, Fields>;
```

Fields

```
/// Set field value
```

```
request<string_body> req;  
req.set("User-Agent", "Chrome");
```

```
/// Inspect field value
```

```
std::cout << "User-Agent:" << req["User-Agent"];
```

Fields

A **Fields** object stores name/value pairs making up fields in an HTTP message.

In this table:

f is a value meeting the requirements of **Fields**

c is a possibly const value meeting the requirements of **Fields**

n,v are values of type `string_view`

Requirements:

Expression	Type	Description
<code>f.set(n,v)</code>		Create a field named <code>n</code> with the value <code>v</code> . If the name already exists, it is overwritten. The name is treated as case-insensitive for comparison.
<code>c[n]</code>	<code>string_view</code>	Returns the value of the previously inserted field named <code>n</code> . If no field exists, an empty string is returned. The field name is treated as case-insensitive for comparison. This function shall not exit via exception.

Fields

```
/// Holds an HTTP response
template<class Body, class Fields>
struct message<false, Body> : Fields
{
    int version;
    int status;

    string reason;

    typename Body::value_type body;
};
```



Fields

```
/// Holds an HTTP response
template<class Body, class Fields>
struct message<false, Body> : Fields
{
    int version;
    int status;

    string_view reason() const;
    void reason(string_view);

    typename Body::value_type body;
};
```



Fields

```
/// Holds an HTTP response
template<class Body, class Fields>
struct message<false, Body> : Fields
{
    int version;
    int status;

    string_view reason() const
    {
        return this->get_reason();
    }

    void reason(string_view s)
    {
        this->set_reason(s);
    }

    typename Body::value_type body;
};
```

Fields

```
/// A associative container for HTTP fields
template<class Allocator>
struct basic_fields
{
    void set(
        string_view name, string_view value);

    string_view operator[](
        string_view name) const;

protected:
    string_view get_reason() const;
    void set_reason(string_view);
};
```

Fields

A **Fields** object stores name/value pairs making up fields in an HTTP message.

In this table:

f is a value meeting the requirements of **Fields**

c is a possibly const value meeting the requirements of **Fields**

n,v are values of type `string_view`

Requirements:

Expression	Type	Description
<code>f.set(n,v)</code>		Create a field named <code>n</code> with the value <code>v</code> . If the name already exists, it is overwritten. The name is treated as case-insensitive for comparison.
<code>c[n]</code>	<code>string_view</code>	Returns the value of the previously inserted field named <code>n</code> . If no such name exists, an empty string is returned. The name is treated as case-insensitive for comparison. This function shall not exit via exception.
<code>c.get_reason()</code>	<code>string_view</code>	Return the previously set reason string of the associated message object.
<code>f.set_reason(v)</code>		Set the reason string of the associated message object to <code>v</code> .

Summary

You're The Boss

Summary

```
/// Holds an HTTP message
template<bool isRequest, class Body, class Fields>
struct message;

/// Holds an HTTP request
template<class Body, class Fields>
struct message<true, Body, Fields>
    : Fields, private Body::value_type
{
    int version;
    string_view method() const
        { return this->get_method(); }
    void method(string_view s)
        { this->set_method(s); }
    string_view target() const
        { return this->get_target(); }
    void target(string_view s)
        { this->set_target(s); }
    typename Body::value_type& body()
        { return *this; }
    typename Body::value_type const& body() const
        { return *this; };
};
```

```
/// A Body that uses a string container
struct string_body
{
    using value_type = string;
    static void read(istream& is, string& body)
        { is >> body; }
    static void write(ostream& os, string const& body)
        { os << body; }
};

// Determine if B meets the requirements of Body
template<class B, class = void>
struct is_body : false_type {};

template<class B>
struct is_body<B, void_t<
    typename B::value_type,
    decltype(
        B::read(
            declval<istream&>(),
            declval<typename B::value_type&>()),
        B::write(
            declval<ostream&>(),
            declval<typename B::value_type const&>()),
            (void)0)
        >> : true_type {};
```

Body Requirements:

Expression	Type	Description
B::value_type		The type of container used to represent the body in a message.
B::read	void(istream&, B::value_type&)	A function invoked by the implementation to parse the body from a std::istream .
B::write	void(ostream&, B::value_type const&)	A function invoked by the implementation to serialize the body to a std::ostream .

Summary

// Code and slides from the talk

<https://github.com/vinniefalco/CppCon2017>

// Boost.Beast library

<https://github.com/boostorg/beast>

// Notes on generic programming

http://www.boost.org/community/generic_programming.html

- Speaker's Dinner

