Techniques in Flexible Header-Only C++ Network Library Implementations

cpp-netlib from the inside

dean.berris@sinefunc.com

About Me

Dean Michael Berris
dean.berris@sinefunc.com

Entrepreneur
Systems Architecture Consultant
C++ Enthusiast

Overview

Rationale

Techniques Used

Going Forward

Rationale

- Build a header-only C++ Network Library
- Implement common protocol clients (and maybe servers) that enable C++ developers to make applications that are network-enabled
- Provide a collection of peer-reviewed implementations
- Foster a community of collaborative development
- Build to one day be part of Boost

Why Header-Only?

To keep it simple to embed in applications that need the functionality.

dean.berris@sinefunc.com

What Protocols To Implement?

HTTP(S) 1.0/1.1
SMTP
FTP
XMPP
ICMP
...

dean.berris@sinefunc.com

Peer Reviewed?

Just like Boost! :)

dean.berris@sinefunc.com

Community?

Working on it. ;)

dean.berris@sinefunc.com

Built to be included in Boost

Because Boost is Cool. :)

dean.berris@sinefunc.com

Techniques Used

Diving into the deep end.

dean.berris@sinefunc.com

Library Organization

3 Major Parts

dean.berris@sinefunc.com

Parts of the Library

- Message mini-framework
  - The basic_message template
  - Directives
    - Transformers
    - Renderers
    - Adapters
- Protocol implementations
  - HTTP 1.0/1.1
    - Client
    - Server
- Utilities and Parsers

dean.berris@sinefunc.com

Technique 1: Common Message Type

Multiple Clients, Multiple Protocols,
One Message: Uniformity

Introducing: basic_message<>

dean.berris@sinefunc.com

Prototype

```
template <class Tag>
struct basic_message;
```

Yes, really -- that's it! :)

dean.berris@sinefunc.com

Usage Semantics

```
basic_message<tag> instance;

source_type s = source(instance);
destination_type d = destination(instance);
headers_type h = headers(instance);
body_type b = body(instance);
```

dean.berris@sinefunc.com

Technique 1.a: Tag-based Design

Almost every type is anchored on a Tag and a metafunction that returns the correct type based on the tag. This means:

```
typedef traits::source<some_tag>::type source_type;

typedef traits::destination<some_tag>::type destination_type;

typedef traits::headers<some_tag>::type headers_type;

typedef traits::body<some_tag>::type body_type;
```

dean.berris@sinefunc.com

You can use this to...

- Define the type of the string or containers to use based on a tag
- Optimize the storage of the basic_message<> based on the tag -- make it a POD, a packed struct (with bit-fields), etc.
- Specialize for the protocol (tags::http_streaming?)
- Write generic code that deals with basic_message template

```
template <class Tag> foo(basic_message<Tag>);
```

dean.berris@sinefunc.com

Technique 1.b: Message Directives

Example:

```
basic_message<tags::default_> message;
message << source("foo")
    << destination("bar")
    << header("foo", "bar")
    << body("w00t");
message << remove_header("foo");
assert(empty(headers(message)["foo"])); // boost.range
```

dean.berris@sinefunc.com

Message Directive Skeleton

```
struct some_directive {
  template <class T> basic_message<T> &
  operator()(basic_message<T> & message) {
    // do something with message
    return message;
  }
  some_directive(foo_type const & instance_data)
  : instance_data(instance_data) {}
  foo_type instance_data;
};
```

dean.berris@sinefunc.com

Message Directive Convenience

```
some_directive some(foo_type const & foo) {
  return some_directive<Tag>(foo);
}

// wired by operator<<
template <class Tag, class Directive>
basic_message<Tag> & operator<<(
    basic_message<Tag> & m,
    Directive const & d
) { return d(m); }
```

dean.berris@sinefunc.com

Message Directive In Action

```
basic_message<tags::default_> message;
message << source("boostcon 2010")
  << some(
        foo_type(
            "constructed out of thin air"
            )
        );
```

dean.berris@sinefunc.com

Technique 2: Semantically Consistent HTTP Client

Syntax -> Structure
Semantics -> Meaning

Introducing: http::client<>

dean.berris@sinefunc.com

Tags, tags, tags...

Just like with the basic_message, we use tags to determine for the HTTP Client:

- What string type to use
- Whether it should be active-async or blocking
- Whether it re-uses connections or keeps it simple
- Whether we support streaming
- Whether we resolve using UDP or TCP
- Whether it throws or not

Table of Tags

Some of these are already supported, while some are still under development (under the tags namespace):

http_default_8bit_(tcp|udp)_resolve
http_keepalive_8bit_(tcp|udp)_resolve
http_async_8bit_(tcp|udp)_resolve
http_stream_8bit_(tcp|udp)_resolve

...

Choosing Your Parents

Instead of explicitly defining policies, we use metafunctions to choose which policies to implement all anchored on the tag.

```
template <class Tag>
struct basic_client :
  policies::resolver<Tag>::type,
  policies::connection<Tag>::type,
  ... {
  ...
};
```

Syntax Mapping Semantics

HTTP Semantics:

1. GET -- retrieve data from a URI
2. PUT -- place data into a URI
3. POST -- add data into a URI
4. DELETE -- delete data associated with a URI

HTTP Client

1. client.get(request) -- retrieve using request
2. client.put(request) -- place data into URI in request
3. client.post(request) -- add data into URI in request
4. client.delete(request) -- delete data in URI in request

Demo: Getting from boost.org

```
cout <<
    body(
        client().get(
            client::request(
                "http://www.boost.org"
                )
            )
        );
```

dean.berris@sinefunc.com

Getting from boost.org... Another way.

```
using namespace boost::network::http;
client client_;
client::request
    request("http://www.boost.org");
client::response
    response_ = client_.get(request);
cout << body(response_);
```

dean.berris@sinefunc.com

Technique 3: Complementing Static and Dynamic Polymorphism

The art of mixing dynamic behavior at compile time and runtime.

dean.berris@sinefunc.com

Strategy Factory: Static

```
template <class Tag>
struct interface {
    virtual void foo() = 0;
    ~interface() { };
};

template <class Tag>
struct strategy {
    unique_ptr<interface<Tag> > create(int input) {/*...*/}
};
```

dean.berris@sinefunc.com

Strategy Factory: Dynamic

```
struct impl1 : interface<Tag> {/* ... */};

struct impl2 : interface<Tag> {/* ... */};

struct impl3 : interface<Tag> {/* ... */};

...

struct implN : interface<Tag> {/* ... */};
```

dean.berris@sinefunc.com

Strategy Factory: Dynamic (continued)

```
template <class Tag>
struct strategy {
    unique_ptr<interface<Tag> > create(int input) {
        unique_ptr<interface<Tag> > impl;
        switch(input) {
            case 1: impl.reset(new impl1()); break;
            case 2: impl.reset(new impl2()); break;
            /*...*/
            default: impl.reset(new implN()); break;
        }
        return impl;
    }
};
```

Static Factory, Dynamic Strategy

Why would you want to do this?

- Retain the static properties of the interface
- Dispatch on runtime values
- Choose the strategy according to runtime values
- Support wiring of variable implementation parts just like with normal OOP

dean.berris@sinefunc.com

Usage in cpp-netlib

To handle HTTPS URI's, there are two types of connections:

- Normal TCP/IP Connection
  - Uses non-encrypted TCP/IP link
  - ASIO sockets are used
- OpenSSL Connection
  - Uses ASIO-provided SSL streams
  - Reaches into underlying socket (encrypted) from the SSL stream.

dean.berris@sinefunc.com

What's in cpp-netlib now?

Features

dean.berris@sinefunc.com

HTTP Client

- HTTP 1.0/1.1 client which manages the sending/receiving of HTTP messages
- Support for HTTP 1.1 Chunked Transfer-Encoding
- Backward-compatible HTTP 1.1 client

```
using namespace boost::network::http;
client client_(client::cache_resolved, client::follow_redirects);
request request_("http://www.boost.org");
response response_;
response_ = client_.get(request_);
response_ = client_.put(request_);
response_ = client_.delete_(request_);
response_ = client_.post(request_);
response_ = client_.head(request_);
```

HTTP Server

```
using namespace boost::network::http;
struct hello;
typedef server<hello> hello_server;

struct hello {
    void operator()(hello_server::request const & req,
                             hello_server::response & res) {
        hello_server::response::stock_reply(
            hello_server::ok, "Hello!"
            );
    }

    void log(...) { }
};
```

dean.berris@sinefunc.com

HTTP Server (continued)

```
int main(int argc, char * argv[]) {
    hello h;
    hello_server server("127.0.0.1", 8080, h);
    server.run();
    return 0;
}

// That's it! :)
```

dean.berris@sinefunc.com

Thank You!

Please don't hesitate to ask questions or approach me!

http://cpp-netlib.github.com/
http://github.com/cpp-netlib/cpp-netlib
http://github.com/mikhailberis/cpp-netlib-boostcon-paper/download

dean.berris@sinefunc.com