A Proposal to the C++ SG4 Working Group for a URI Library

Glyn Matthews[1], Dean Michael Berris[2]

[1] glyn.matthews@gmail.com
[2] dberris@google.com

Introduction

The library proposal I will make in this document is based on the URI in the project *C++ Network Library (cpp-netlib)*[3]. This library is still under development and subject to change itself, particularly as we start to add support for C++11 features. Therefore, what we propose is an "ideal" library based on this implementation, that includes C++11 features and other desirable features that aren't yet implemented.

History

The *cpp-netlib* project was founded in 2007 in order to develop a network library using modern C++ techniques using Boost.Asio as its base. Since then the project has grown to include a fully compliant HTTP client and server, and a class that represents URIs.

Implementation

The working implementation of the *cpp-netlib* `uri` class currently implements RFC 3986[4] and RFC 2732[5].

Headers

```
#include <network/uri>
```

Namespaces

```
namespace std::network;
```

URI Class Description

```
class uri
{
public:

    typedef char value_type;
    typedef std::string string_type;
    typedef std::string::const_iterator iterator;
    typedef std::pair<string_type::const_iterator,
                      string_type::const_iterator> iterator_range;

    typedef std::codecvt<...> wstring_codecvt_t;
    typedef std::codecvt<...> u16string_codecvt_t;
    typedef std::codecvt<...> u32string_codecvt_t;

    uri();
    uri(const string_type &uri_string);
    template <Iterator> uri(Iterator first, Iterator last);
    uri(const uri &other);
    uri(uri &&other);
```

---

[3] http://cpp-netlib.github.com/

[4] http://tools.ietf.org/html/rfc3986

[5] http://tools.ietf.org/html/rfc2732

```
    uri &operator = (const string_type &uri_string);
    uri &operator = (uri other);
    ~uri();

    // URI checks
    bool valid() const;
    bool hierarchical() const;
    bool opaque() const;
    bool relative() const;
    bool absolute() const;

    // range accessors
    iterator_range scheme_range() const;
    iterator_range user_info_range() const;
    iterator_range host_range() const;
    iterator_range port_range() const;
    iterator_range path_range() const;
    iterator_range query_range() const;
    iterator_range fragment_range() const;

    // string accessors
    string_type string() const;
    std::wstring wstring(const wstring_codecvt_t &cvt =
                         wstring_codecvt_t()) const;
    std::u16string u16string(const u16string_codecvt_t &cvt =
                             u16string_codecvt_t()) const;
    std::u32string u32string(const u32string_codecvt_t &cvt =
                             u32string_codecvt_t()) const;
    string_type scheme() const;
    string_type user_info() const;
    string_type host() const;
    string_type port() const;
    string_type path() const;
    string_type query() const;
    string_type fragment() const;

    void swap(uri &other) noexcept;

};
```

The `uri` class itself, is a little more than a light-weight wrapper around a string, a parser and the `uri`'s component parts. Parsing is performed upon construction and, if successfully parsed, the component parts are stored as iterator ranges that reference the original string.
For example, consider the following URI:

```
http://www.example.com/path/?key=value#fragment
^   ^  ^              ^     ^         ^^       ^
a   b  c              d     e         fg       h
```

On parsing, the `uri` object will contain a set of iterator pairs corresponding to the ranges for *scheme*, *user info*, *host*, *port*, *path*, *query* and *fragment*. So the pairs of iterators corresponding to the example above will be:

URI part	**Range**	**String**
scheme	`[a, b)`	"http"
user_info	`[c, c)`	""
host	`[c, d)`	"www.example.com"
port	`[d, d)`	""
path	`[d, e)`	"/path/"
query	`[e, f)`	"?key=value"
fragment	`[g, h)`	"fragment"

One advantage of this is that the memory footprint of the `uri` remains small. A second, is that the URI interface can be more easily extended using these iterator ranges.

URI Operators

```
bool operator < (const uri &lhs, const uri &rhs);
bool operator == (const uri &lhs, const uri &rhs);
bool operator == (const uri::string_type &lhs, const uri &rhs);
bool operator == (const uri &lhs, const uri::string_type &rhs);
bool operator == (const uri::value_type *lhs, const uri &rhs);
bool operator == (const uri &lhs, const uri::value_type *rhs);
std::ostream &operator << (std::ostream &os, const uri &uri_);
```

The `uri` class is less-than comparable and equality comparable.

URI Operational Functions

```
void swap(uri &lhs, uri &rhs);
```

The `uri` is swappable.

```
std::size_t hash_value(const uri &uri_);
```

The uri is hashable, meaning that it can be used as a key in an `std::unordered_map` and `std::unordered_set`.

```
uri from_parts(const uri::string_type &base,
               const uri::string_type &path);
uri from_parts(const uri::string_type &base,
               const uri::string_type &path,
```

```
                const uri::string_type &query);
uri from_parts(const uri::string_type &base,
                const uri::string_type &path,
                const uri::string_type &query,
                const uri::string_type &fragment);
uri from_parts(const uri &base,
                const uri::string_type &path);
uri from_parts(const uri &base,
                const uri::string_type &path,
                const uri::string_type &query);
uri from_parts(const uri &base,
                const uri::string_type &path,
                const uri::string_type &query,
                const uri::string_type &fragment);
uri from_file(const boost::filesystem::path &path);
```

A `uri` object can be constructed using these convenience functions from its component parts, or from common types, such as a filesystem path.

```
uri::string_type scheme(const uri &uri_);
uri::string_type user_info(const uri &uri_);
uri::string_type host(const uri &uri_);
uri::string_type port(const uri &uri_);
std::uint16_t port_us(const uri &uri_);
uri::string_type path(const uri &uri_);
uri::string_type decoded_path(const uri &uri_);
uri::string_type query(const uri &uri_);
uri::string_type fragment(const uri &uri_);
```

These free functions are simple accessors to the `uri` object. The advantage of this style of interface is the way in which they can be extended. As mentioned previously, the `uri` object provides a set of ranges around the URI parts. Using these parts, it's possible to extend the uri interface efficiently.

A motivating example is to that it might be necessary to access the whole hierarchical part of a URI. This can be implemented as follows:

```
uri::string_type hierarchical_part(const uri &uri_) {
    return uri::string_type(std::begin(uri_.user_info_range()),
                            std::end(uri_.path_range()));
}
```

A second motivating example would be that it would be convenient to add an accessor to get only the `username` from the `user_info` part:

```
uri::string_type username(const uri &uri_) {
    auto user_info = uri_.user_info_range();
    auto colon = std::find_if(std::begin(user_info), std::end(user_info),
                              [] (uri::value_type v) { return v == ':'; });
```

```
    return uri::string_type(std::begin(user_info), colon);
}
```

URI Builder Class Description

```
class builder
{
    builder(const builder &) = delete;
    builder &operator = (const builder &) = delete;

public:

    explicit builder(uri &uri_);
    ~builder();

    builder &scheme(const string_type &scheme);
    builder &user_info(const string_type &user_info);
    builder &host(const string_type &host);
    builder &host(const boost::asio::ip::address &host);
    builder &host(const boost::asio::ip::address_v4 &host);
    builder &host(const boost::asio::ip::address_v6 &host);
    builder &port(const string_type &port);
    builder &port(std::uint16_t port);
    builder &path(const string_type &path);
    builder &encoded_path(const string_type &path);
    builder &query(const string_type &query);
    builder &query(const string_type &key,
                   const string_type &value);
    builder &fragment(const string_type &fragment);

};
```

The builder is intended to allow a `uri` object to be created more consistently from its component parts.

Examples

Listed below are some simple examples of the `uri` in use:

Construction from a string

```
network::uri uri_(“http://www.example.com/”);
assert(uri_.is_valid());
assert(“http://www.example.com/” == uri_);
```

Construction from a pair of iterators

```
std::wstring uri_string(L“http://www.example.com/”);
network::uri uri_(std::begin(uri_string), std::end(uri_string));
assert(uri_.is_valid());
assert(“http://www.example.com/” == uri_);
```

Accessing URI parts

```
network::uri uri_(“http://www.example.com/path/to/resource/”);
assert(uri_.is_valid());
assert(“http” == network::scheme(uri_));
assert(“www.example.com” == network::host(uri_));
assert(“/path/to/resource/” == network::path(uri_));
```

Building a URI from its parts

```
network::uri base_uri_(“http://www.example.com”);
network::uri uri_(network::from_parts(base_uri_, “/path/to/resource/”);
assert(“http://www.example.com/path/to/resource/”, uri_);
assert(“http” == network::scheme(uri_));
assert(“www.example.com” == network::host(uri_));
assert(“/path/to/resource/” == network::path(uri_));
```

Building a URI from a filesystem path

```
boost::filesystem::path path_(“/path/to/a/file.txt”);
network::uri uri_(network::from_file(path_));
assert(“file:///path/to/a/file.txt” == uri_);
assert(“file” == network::scheme(uri_));
assert(“/path/to/a/file.txt” == network::path(uri_));
```

Building a URI query

```
network::uri uri_;
network::builder(uri_);
builder
    .scheme(“http”)
    .host(“www.example.com”)
    .path(“/”)
    .query(“key1”, “value1”)
    .query(“key2”, “value2”)
    ;
assert(“http://www.example.com/?key1=value1&key2=value2” == uri_);
assert(“?key1=value1&key2=value2” == network::query(uri_));
```

Decoding a URI path

```
network::uri uri_(“http://en.wikipedia.org/wiki/C%2B%2B”);
assert(“/wiki/C++” == network::decoded_path(uri_));
assert(“/wiki/C++” == network::decode(network::path(uri_)));
```

Relationship to `std::string` and `boost::filesystem::path`[6]

The interface design is strongly influenced by both `std::string` and `boost::filesystem::path` and, as looks likely, its future standard equivalent. To be successful, the `uri` class will need to complement these classes.

A Note on the Parser

[6] http://www.boost.org/libs/filesystem/

The current implementation of cpp-netlib's `uri` parser uses Boost.Spirit[7]. This proposal will recommend that the parser be *implementation-defined*, but should comply at least with RFC 3896 and RFC 2732. Using only standard library components, a reasonable implementation might use `std::regex`, but beyond this there are far too requirements to consider, including parser efficiency and non-standard extensions (for example, the support of Windows filesystem path separators).

A Note on Dependencies

The `network::uri` class depends on `std::string` and a limited set of Boost libraries. Some of these may be proposed for inclusion for C++1y. These include:

Boost.Optional[8]
Boost.Asio[9]
Boost.Range[10]

Conclusion

The `uri` class that is proposed in this document takes advantage of C++'s iterator and range concepts for efficiency and extensibility with a minimalist interface. It's style is consistent with existing C++ libraries, and can leverage C++11 features (particularly string literals) to be usable across a wide range of existing C++ applications as possible.

Future Direction

Support for internationalized URIs (RFC 3987[11]).
Add a `network::uri::parse_exception` instead of requiring that the caller checks `network::uri::valid()`.
Add a `normalize_path` member function.

References

http://cpp-netlib.github.com/in_depth/uri.html

[7] http://www.boost.org/libs/spirit/
[8] http://www.boost.org/libs/optional/
[9] http://www.boost.org/libs/asio/
[10] http://www.boost.org/libs/range/
[11] http://tools.ietf.org/html/rfc2732