RFC822 Address Parser Library 1.0

Peter Simons

simons@computer.org

Table of Contents

1. Purpose of this Library

librfc822 provides application developers with a complete parser for RFC822 addresses. Not only can you use the library to verify that RFC822 addresses are syntactically correct, you can also have an address split up into its semantic parts, what is needed when deciding where to route an address to, etc.

What is quite unique is that librfc822 does indeed parse *all* address types allowed by the standard. That includes such weird things as “address groups” or addresses with whitespace and comments throw in. Take a look at this beast to get an idea:

```
testing my parser : peter.simons@gmd.de,
        (peter.)simons@rhein.de „„,
     testing my parser <simons@ieee.org>,
     it rules <@peti.gmd.de,@listserv.gmd.de:simons @ cys .de>
     ;
     ,
     peter.simons@acm.org
```

That is indeed a legal e-mail address in RFC822 messages. It contains five separate addresses, which are grouped together. Here’s the parsed result:

```
peter.simons@gmd.de
simons@rhein.de
simons@ieee.org
<@peti.gmd.de,@listserv.gmd.de:simons@cys.de>
peter.simons@acm.org
simons@rhein.de
simons@rhein.de
simons@rhein.de
```

In case you wonder: The strange looking address that’s listed fourth is a so called “routing address” – and yes, that’s a legal e-mail address, too. These were popular in the early days of the Internet. Back then, every mail server that relayed an e-mail put its own address into this construct so that bounces could be routed the same way back that they originally went. The address says that the mail should be send to the host `peti.gmd.de`, then to `listserv.gmd.de`, and from there it should be delivered (using any route) to the address `simons@cys.de`. These days, such addresses can hardly be used, because nobody relays for other recipients anymore. Still, these are legal.

librfc822 provides you with several routines that parse the different flavours of e-mail addresses as defined in the standard. The results will be placed in a rfc822address structure and returned. If constructs are parsed that may contain multiple addresses, you can pass a “committer” class to the function, which is called every time a correct address is found and may append it to a container of your choice.

2. The high-level Interface

You can use librfc822's internal parser class directly, if you need complete control over everything, but that's not for the faint hearted. Honestly, I use the high-level interface myself! "High-level" means that the include `rfc822.hh` defines a set of routines which parse a certain type of address each. In contrast, accessing the parser directly would give you the ability to parse arbitrary RFC822 headers with relatively little effort. Eventually I will provide comfortable routines for this purpose, too; that's on the "to do" list.

2.1. The rfc822address Structure

When dealing with librfc822, the results of the parsing process will be placed into the rfc822address structure, which is defined as follows:

```
 class rfc822address {
  std::string address;
  std::string localpart;
  std::string hostpart;
}
```

The *`address`* field will contain the *complete* address without any comments, whitespace, or whatever madness the standard allows. If you want to compare to addresses for equity, this is the place to go. The *`localpart`* field is set to the localpart of the address. "localpart" in that context does not necessarily mean "username". For addresses of the type "user@example.org", "user" will be the localpart, granted, but when parsing the routing address "<@example.com:user@example.org>", the localpart will be "user@example.org". Imagine the "localpart" as being that part of the address that remains when the "hostpart" is stripped off!

In case of a routing address, the hostpart will be the first hostname in the list of routed hosts and the remainder will be the localpart. So, when having an address parsed by the library, you can check whether the address is really a local address by checking whether the *`localpart`* field does still contain an "@". If it does, parse it again (and again, and again ...) to get the username.

The *`hostpart`* field will contain the name of the host that would interpret this address as "local". This should be relatively clear from the discussion of the *`localpart`* field.

`rfc822.hh` defines an `operator«` for rfc822address, so you can print instances of the structure to any std::ostream, but the format printed by the operator is meant more or less for debugging purposes, not for anything useful. An `operator»` is not defined by the library.

2.2. The `check_rfc822_`*`xxx`* functions

This set of free functions is defined in `rfc822.hh`. Each of these routines parses a certain type of address, as defined in the standard, but no results are returned. Rather, these functions will be used to verify the addresses syntax. If an address contains an error, an rfc822_syntax_error exception will be thrown.

The available routines are:

`void` **`check_rfc822_addr_spec`**`(const std::string` *`input`*`);`

> Verify an address of the form: `local-part "@" domain`.

`void` **`check_rfc822_route_addr`**`(const std::string` *`input`*`);`

> Verify an address of the form: `"<" [route] addr-spec ">"`.

`void` **`check_rfc822_mailbox`**`(const std::string` *`input`*`);`

> Verify an address of the form: `addr-spec | phrase route-addr`.

`void` **`check_rfc822_address`**`(const std::string` *`input`*`);`

> Verify an address of the form: `mailbox | group`.

`void` **`check_rfc822_mailboxes`**`(const std::string` *`input`*`);`

> Verify an address of the form: `mailbox ("," mailbox)*`.

`void` **`check_rfc822_addresses`**`(const std::string` *`input`*`);`

> Verify an address of the form: `address ("," address)*`.

Obviously, the syntax specification here is a bit short. Please refer to section 6 of [RFC822] for further details!

2.3. The `parse_rfc822_xxx` functions

This set of free functions is defined in `rfc822.hh`. Each of these routines parses a certain type of address, as defined in the standard, and returns the result in one or more rfc822address structure. The routines that parse a *single* address will return the structure directly as the return value, the routines that may possibly return *multiple* addresses require an additional parameter of type std::insert_iterator<T>* – a pointer to a class instance that will be used to append the results to a container of your choice.

If an address contains an syntax error, an rfc822_syntax_error exception will be thrown. Please note that when the exception is thrown, an arbitrary number of addresses might already have been added to the container!

The available routines are:

`rfc822address` **`parse_rfc822_addr_spec`**`(const std::string` *`input`*`);`

> Parse an address of the form: `local-part "@" domain`.

`rfc822address` **`parse_rfc822_route_addr`**`(const std::string` *`input`*`);`

> Parse an address of the form: `"<" [route] addr-spec ">"`.

`rfc822address` **`parse_rfc822_mailbox`**`(const std::string` *`input`*`);`

> Parse an address of the form: `addr-spec | phrase route-addr`.

`void` **`parse_rfc822_address`**`(std::insert_iterator<T>*` *`ii`*`, const std::string` *`input`*`);`

> Parse an address of the form: `mailbox | group`.

`void` **`parse_rfc822_mailboxes`**`(std::insert_iterator<T>*` *`ii`*`, const std::string` *`input`*`);`

> Parse an address of the form: `mailbox ("," mailbox)*`.

`void` **`parse_rfc822_addresses`**`(std::insert_iterator<T>*` *`ii`*`, const std::string` *`input`*`);`

> Parse an address of the form: `address ("," address)*`.

Obviously, the syntax specification here is a bit short. Please refer to section 6 of [RFC822] for further details!

One more note: It is legal to pass “0” for the *`ii`* parameter; in that case, the routines will throw all parsers results away and effectively act like the corresponding `check_rfc822_xxx` function.

3. The low-level Interface

If don't trust those comfortable high-level routines, you may access the RFC parser directly. This will also allow you to re-use parts of the parser to implement new parsers for other RFC822 components or for parsers that need to understand a similar syntax.

3.1. The Lexer

The lexer used in librfc822 is a free function called `lex`, which is defined in `rfc822.hh`. Its synopsis is:

```
tokstream_t lex(const std::string& input);
```

`lex` will walk throught the text buffer *`input`* and produce a stream of actual tokens. Everything that's *not* a token according to the standard is omitted, such as whitespace, comments, etc. tokstream_t is defined to be std::deque<token> in `rfc822.hh`, but don't depend on the choice of the container, because it might change in future versions of the library. Use the abstract name instead.

The `token` structure contains an enumerator, which defines the type of the token and the actual string value of the token. Please refer to `rfc822.hh` and `lexer.cc` if you want to know those details. They are usually not important for users of the library.

3.2. The Parser

The parser of librfc822 consists of the class `rfc822parser`, which is defined in `rfc822.hh`. It's public interface is:

```
 class rfc822parser {
  rfc822parser(tokstream_t ts, address_comitter* c);
  void addresses();
  void mailboxes();
  void address();
  rfc822address mailbox();
  rfc822address route_addr();
  rfc822address addr_spec();
  bool empty();
}
```

Obviously, the parser class must be created with a token stream as returned by `lex` and a pointer to a committer class, which is used to return the parsed results by the member functions that may return multiple addresses. This data is stored internally in the class and may be erased once `rfc822parser` is instantiated. Consequently, you may parse only one token stream per parser instance, but creating an instance is not very expensive.

Once the `rfc822parser` instance has been created, you may call the various member functions repeatedly to parse the corresponding expression. The parsed tokens are thereby consumed from the internal token stream. If no tokens are left, the `rfc822parser::empty()` function will return `true`.

3.3. The Address Committer

The class `rfc822parser::address_committer` defines an interface from which you can derive your own committer classes. `rfc822.hh` defines this interface as follows:

```
 class rfc822parser::address_committer {
  virtual void operator()(const rfc822address& address);
}
```

It effectively implements a simple callback. Whenever any of the member functions `rfc822parser::address`, `rfc822parser::addresses`, or `rfc822parser::mailboxes` finds a complete RFC822 address, it will invoke the instance of the committer class that `rfc822parser` has been instantiated with. If the class has been instantiated with “0” for a pointer to the committer class, the results of the parser will be thrown away.

3.4. An Example Program

```
#include <rfc822.hh>
using namespace std;

class my_committer : public rfc822parser::address_committer
    {
  public:
    void operator() (const rfc822address & addr)
        {
        cout « addr « endl;
        }
    };

int main()
try
    {
    string input = \
        "testing my parser : peter.simons@gmd.de,\n"    \
        "\t (peter.)simons@rhein.de „„,\n"            \
        "\t testing my parser <simons@ieee.org>,\n"     \
        "\t it rules <@peti.gmd.de:simons @ cys .de>\n" \
        "\t ;\n"                                        \
        "\t ,\n"                                        \
        "\t peter.simons@acm.org\n";

    my_committer   committer;
    rfc822parser parser(lex(input), &committer);
    parser.addresses();

    return 0;
    }
catch(rfc822_syntax_error & e)
    {
    cout « "Address contains an syntax error: " « e.what() « endl;
    }
catch(...)
    {
    cout « "Caught unknown exception." « endl;
    }
```

4. Exceptions Thrown by librfc822

The only exception actually thrown by librfc822 is the rfc822_syntax_error, which occurs in case of an syntax error. Other exceptions may be throw by the classes used in the library, that's your problem. :-)

The class is defined as follows:

```
 class rfc822_syntax_error : public std::runtime_error {
  virtual const char* what();
}
```

The `what` member function inherited from std::runtime_error provides you with a text description of the syntax error that caused this exception to be thrown.

Arguably this interface doesn't provide the programmer with overly detailed information for error recovery and I always wanted to polish it, but the truth is: I have used librfc822 in several programs of mine and never actually *needed* something else! That's why the library still provides only this simple mechanism for error reporting.

5. License

This software is copyrighted by Peter Simons <`simons@computer.org`>. Permission is granted to use it under the terms of the GNU General Public License. For further details, refer to the file `LICENSE` included in the software distribution or see http://www.gnu.org/licenses/gpl.html in case that file is missing.

Bibliography

[RFC822] David H. Crocker, *Request for Comments 822: “Standard for the Format of ARPA Internet Text Messages” (http://rfc.fh-koeln.de/rfc/html_gz/rfc822.html.gz)* .