Efficient Sequences from Streams

Jeff Trull

13 May 2015

Inspiration
Why this interests me

- Sequence processing in scripting languages: map, reduce, zip etc.
- S. Parent, 2013, “C++ Seasoning” and eliminating raw loops
- E. Niebler, Range-v3
- K. Ahnert, 2014, “Ranges and iterators for numerical problems”
- Lazy evaluation in general

Most examples start with the data already in containers... what if I don’t want to (or can’t) store everything in advance?

std::istream_iterator

Some support for input sequences is built in to the language already:

```
template<typename T>
void separated_printer(std::string const& s) {
    typedef std::istream_iterator<T> iter_t;
    std::istringstream ss(s);
    for (auto it = iter_t(ss); it != iter_t(); ++it) {
        std::cout << "|" << *it << "| ";
    }
    std::cout << "\n";
}
int main() {
    std::string test("123 ABC");
    separated_printer<char>(test);        // |1| |2| |3|
        |A| |B| |C|
    separated_printer<std::string>(test); // |123| |ABC|
    separated_printer<int>(test);         // |123|

}
```

ranges::v3::istream_range

You can directly port std::istream_iterator-based code to range-v3:

```
template<typename T>
void separated_printer(std::string const& s) {
    using namespace ranges::v3;
    std::istringstream ss(s);
    ranges::for_each(istream_range<T>(ss),
                     [](T const& v) {
                         std::cout << "|" << v << "| ";
                     });
    std::cout << "\n";
}
```

A More Complex Example

NMEA 0183 is a protocol standard for marine electronics that is commonly used by GPS receivers. It consists of a series of “sentences” giving useful status information, and looks like this:

```
$GPGSV,3,3,11,29,09,301,24,16,09,020,,36,,,*76
$GPRMC,092750.000,A,5321.6802,N,00630.3372,W,0.02,31.66,280511,,,A*43
$GPGGA,092751.000,5321.6802,N,00630.3371,W,1,8,1.03,61.7,M,55.3,M,,*75
```

(example taken from Wikipedia)

$GPGGA

This is the “GPS Fix Data” message **$GPGGA**

```
struct gga {
    int                   timestamp;
    double                latitude;
    char                  lat_hemi;
    double                longitude;
    char                  long_hemi;
    int                   quality;
    int                   sat_count;
    double                dilution;
    double                altitude;
    char                  alt_units;
    double                sea_level;
    char                  sea_level_units;
    boost::optional<int>  time_since_dgps;
    boost::optional<int>  dgps_station_id;
    int                   checksum;
};
```

Custom Stream Operator

With a custom input stream operator we can reuse our previous approach

First, a quick helper function:

```
std::istream& comma(std::istream& is) {
    char ch;
    is >> ch;
    if (ch != ',') {
        is.setstate(std::ios_base::failbit);
    }
    return is;
}
```

Custom Stream Operator

With a custom input stream operator we can reuse our previous approach

Now the operator itself:

```
std::istream&
operator>>(std::istream& is, gga& g) {
    char ch;
    string str;

    while (!is.fail() && str.length() < 6) {
        is >> ch;
        str += ch;
    }
    if (str != "$GPGGA") {
        is.setstate(ios_base::failbit);
    }
    comma(is);

    is >> g.timestamp >> comma >> g.latitude >> comma;
```

Custom Stream Operator

With a custom input stream operator we can reuse our previous approach

```
    is >> g.lat_hemi;
    if ((g.lat_hemi != 'N') && (g.lat_hemi != 'S')) {
        is.setstate(ios_base::failbit);
    }
    comma(is);

    // handle optional values
    if (is.peek() != ',') {
        is >> g.time_since_dgps;
    }
    comma(is);

    // etc...
```

50 lines of code to stream in **$GPGGA**?

Using Our New Range

Producing a filtered range of Boost.Geometry world coordinates

```
using namespace boost::geometry;
typedef model::point<double, 2,
                     cs::geographic<degree>> coord_t;
auto coord_range =
    istream_range<gga_t>(ss) |
    ranges::view::remove_if([](gga_t const& g) {
            return g.dilution > 1.0;
        }) |
    ranges::view::transform([](gga_t const& g) {
            double lat_deg = floor(g.latitude / 100.0);
            if (g.lat_hemi == 'W') {
                lat_deg = -lat_deg;
            }
            double long_deg = floor(g.longitude / 100.0);
            if (g.lat_hemi == 'S') {
                long_deg = -long_deg;
            }
            return coord_t(lat_deg, long_deg);
        });
```

That Was a Bit Painful

We should use a proper parser

Spirit has a "Stream based parse API" that adapts parsers to iostream manipulators. First, let's define a grammar:

```
std::istream&
operator>>(std::istream& is, gga_t& g) {
    using namespace boost::spirit;

    qi::rule<istream_iterator, gga_t()> gga_parser =
        qi::lit("$GPGGA") >> ',' >> qi::int_ >> ',' >>
        qi::double_ >> ',' >> qi::char_("NS") >> ',' >>
        qi::double_ >> ',' >> qi::char_("EW") >> ',' >>
        qi::int_ >> ',' >> qi::int_ >> ',' >>
        qi::double_ >> ',' >>
        qi::double_ >> ',' >> qi::char_ >> ',' >>
        qi::double_ >> ',' >> qi::char_ >> ',' >>
        -qi::int_ >> ',' >> -qi::int_ >> ",*" >> qi::int_
            ;

    // construct a stream manipulator from this parser
    auto manip = qi::match(gga_parser, g);
    is >> manip;
    return is;
}
```

Using the Range Facade

A parser wrapped in a manipulator adapter wrapped in a stream operator wrapped in a stream range... Let's cut out some of these layers:

```
class gga_range_t : public ranges::range_facade<gga_range_t>
{
    // for facade access:
    friend ranges::range_access;
    struct cursor;
 public:
    typedef boost::spirit::istream_iterator iter_t;
    gga_range_t(std::istream& is) :
            m_stream(&is), m_it(*m_stream), m_parse_good(true) {
        m_parser = ... // define GPGGA rule
    }
    cursor begin_cursor() {
        return cursor(*this);
    }
private:
    void next() {
        m_parse_good = boost::spirit::qi::parse(m_it, iter_t(), m_parser,
             m_g);
    }
    gga_t cached() const { return m_g; }
    bool done() const { return !m_parse_good || !m_stream; }
    ...
};
```

Using the Range Facade

The cursor keeps a pointer to the parent class, which is uncopyable... The range is built from the cursor by the facade class:

```
class gga_range_t : public ranges::range_facade<gga_range_t>
{
    ...
    struct cursor {
        cursor(void) = default;
        explicit cursor(gga_range_t& rng) : m_rng(&rng) {
            next();     // to load the first value
        }
        void next() {
            m_rng->next();
        }
        gga_t current() const {
            return m_rng->cached();
        }
        bool done() const {
            return m_rng->done();
        }
        gga_range_t* m_rng;
    };
};
```

Working with a full NMEA grammar
My scheme breaks

NMEA has several different kinds of “sentences” that may appear in any order. Simply attempting to pull a GPGGA from the input each time won’t work. Moreover, in general, inputs are not structured as a sequence of identical items. We need to parse the full language to know what parts belong in our sequence. However:

- The resulting code is no longer lazy
- Memory usage could potentially be quite large
- We cannot handle infinite sequences

We need a parser that stops and emits each of our values when it finds them, while retaining its state.

Spirit + Coroutine

You saw this coming, didn't you...

```
typedef boost::spirit::istream_iterator iter_t;
using namespace boost;
struct nmea_parser : spirit::qi::grammar<iter_t>
{
    typedef coroutines::asymmetric_coroutine<gga_t>::push_type sink_t;
    nmea_parser(sink_t& sink)
        : nmea_parser::base_type(sentences), m_sink(sink)
    {
        using namespace spirit;
        namespace phx = phoenix;

        gga_sentence = ... // the usual
        // define other "sentences" here
        sentences = *(gga_sentence   // | bwc_sentence | gll_sentence | ...
             )
                        [phx::bind(&sink_t::operator(),
                                   phx::ref(m_sink), qi::_1)]
                       >> -qi::eol);
    }
    qi::rule<istream_iterator, gga_t()> gga_sentence;
    qi::rule<istream_iterator> sentences;

    // destination for GGA sentences as they are parsed
    sink_t& m_sink;

};
```

Spirit + Coroutine
Usage

The Coroutine library generates a sequence for us automatically:

```
using namespace boost::coroutines;
asymmetric_coroutine<gga_t>::pull_type sequence(
    [&ss](asymmetric_coroutine<gga_t>::push_type&
        sink) {
        nmea_parser nmea(sink);
        iter_t beg(ss), end;
        if (!boost::spirit::qi::parse(beg, end, nmea)
            ) {
            std::cerr << "parse failed!\n";
        }
    });

for (gga_t const& g : sequence) {
    std::cout << g << "\n";
}
```

Resources

- The Range v3 library
  `https://ericniebler.github.io/range-v3/index.html`
- Karsten Ahnert on ranges and numerical problems
  `https://meetingcpp.com/index.php/br/items/ranges-for-numerical-problems-402.html`
- All code samples from this talk
  `https://github.com/jefftrull/SequencesFromStreams`