BoostCon 2011

Rob Stewart

Spirit.Qi in the Real World

- Why Spirit?
- Initial difficulties
- The example

Overview

Why Spirit?

- Spirit permits creating PEG grammars within C++
  - No extra tools
  - No extra build steps
  - Powerful, capable
- Semantic actions, written with Phoenix, even look like C++

Why Spirit?

Initial Difficulties

Initial Difficulties

- Good documentation
  - Illustrates many aspects of Spirit
  - Presents many advanced topics
- More documentation needed
  - Index just appeared in Boost 1.46
  - TOC doesn't indicate concepts covered by examples
  - Seems to stop just where information needed
  - No comparisons of options

Initial Difficulties

- No documentation on debugging
- No explanation of likely compiler errors and their meaning
  - Good use of static assertions with comments to highlight matters
  - Typical advice
    - Search backward in error trace for *****
    - Look in corresponding Spirit header
    - Likely find a helpful comment
  - Also techniques depending upon compiler

The Example: printf()

- Full POSIX format support
- Developed from simplest to complete
- Debugging support
- Useful diagnostics

The Example: printf() Format Parser

- Some familiarity with Spirit.Qi v2
- Familiar with `printf()`

Assumptions

- No output
- No conversions
- Just format parsing

Simplifications

- Ordinary characters (not %)
- %%
- Conversion specifications
  - %
  - Flag characters (zero or more)
  - Field width (optional)
  - Precision (optional)
  - Length modifier (optional)
  - Conversion specifier

printf() Format Strings

- Parse then convert
- Convert while parsing

Two Basic Approaches

Parse Then Convert

- Parse full format string
  - Save normal text
  - Save conversion specification state and position
- Convert
  - Write normal text to output until need converted value
  - Convert argument based upon saved state
  - Repeat

- Write normal text to output
- Parse one conversion specification
- Convert one argument
- Repeat

Convert While Parsing

- Pass through normal text
- Look for %'s
- Look for optional parts
- Look for the conversion specifier

Same Parsing Needed

- Start simple
- Test
- Add complexity stepwise
- Test with each addition
- Add diagnostics

Writing a Parser

Code Simplifications

```
namespace phx = boost::phoenix;
namespace qi = boost::spririt::qi;
using qi::char_;
using qi::lit;
using qi::_val;
using qi::_1;
using phx::val;
using phx::ref;
```

Parsing %d

- `"%d"`
- `lit('%') >> 'd';`
- That recognizes the input, but what next?
- Save it
- Need something to save a conversion specification's state

conversion_specifier

```
enum conversion_specifier
{
   CS_CHARACTER,   // %c
   CS_DECIMAL,     // %d, %u
   CS_FIXED,       // %f, %F
   CS_GENERAL,     // %g, %G
   CS_HEXADECIMAL, // %x, %X
   CS_OCTAL,       // %o
   CS_POINTER,     // %p
   CS_SCIENTIFIC,  // %e, %E
   CS_STRING,      // %s
   CS_UNSIGNED,    // %u
   CS_WRITTEN      // %n
};
```

Capturing the Specifier

- Make a rule with a `conversion_specifier` synthesized attribute

```
qi::rule<It,conversion_specifier()> specifier;
specifier = lit('%') >> 'd';
```

- Set the synthesized attribute

```
specifier
    = lit('%')
    >> lit('d')[_val = val(CS_DECIMAL)]
    ;
```

Capturing the Specifiers

- Need to support the other conversion specifiers
- Several means to the end
  - Alternation
  - `qi::symbols`
  - Phoenix function
  - Phoenix lambda

Capturing the Specifiers: Alternation

```
specifier
   = lit('%')
   >>
      (
           lit('d')[_val = val(CS_DECIMAL)]
         | lit('f')[_val = val(CS_FIXED)]
         | lit('g')[_val = val(CS_GENERAL)]
         ...
      )
   ;
```

```
"%d"
"%f"
"%g"
```

```
specifier
   = lit('%')
   >> char_("cdEeFfGginopsuXx")
      [_val = get_specifier(_1)]
   ;
```

```
"%c"
"%d"
"%E"
...
```

- `get_specifier` translates parsed character to enumerator
- Declare Phoenix function

```
phx::function<get_specifier_impl> get_specifier;
```

- Namespace scope or grammar data member

Capturing the Specifiers: Phoenix Function

get_specifier_impl

```
struct get_specifier_impl
{
   template <class Char>
   conversion_specifier
   operator ()(Char _specifier) const
   {
      switch (_specifier)
      {
         char 'd': return CS_DECIMAL;
         char 'f': return CS_FIXED;
         char 'g': return CS_GENERAL;
         ...
      }
   }
};
```

"%d"
"%f"
"%g"

```
struct get_specifier_impl
{
   template <class>
   struct result
   {
      typedef conversion_specifier type;
   };

   template <class Char>
   typename result<Char>::type
   operator ()(Char _specifier) const
   {
      ...
   }
};
```

get_specifier_impl Result Type

Capturing the Specifier: Phoenix Function

```
qi::rule<It,conversion_specifier()> specifier;
specifier
   = lit('%')
   >> char_("cdEeFfGginopsuXx")
      [_val = get_specifier(_1)]
   ;
```

Capturing the Specifier: qi::symbols

```
qi::symbols<char,conversion_specifier> specifiers;
specifiers.add
   ("d", CS_DECIMAL)
   ("f", CS_FIXED)
   ("g", CS_GENERAL)
   ...
   ;
specifier = lit('%') >> specifiers;
```

```
"%d"
"%f"
"%g"
```

Capturing the Specifier: Phoenix Lambda

```
specifier
   = lit('%')
   >> char_("cdEeFfGginopsuXx")
      [
         phx::switch_(_1)
         [
            phx::case_<'d'>(_val = val(CS_DECIMAL)),
            phx::case_<'f'>(_val = val(CS_FIXED)),
            phx::case_<'g'>(_val = val(CS_GENERAL)),
            ...
         ]
      ]
   ;
```

"%d"
"%f"
"%g"

Capturing the Specifier: Summary

- Four techniques:
  - Alternation
  - `qi::symbols`
  - Phoenix function
  - Phoenix lambda
- Different compile time demands
- Different locality of reference

- More Phoenix implies more compile time
  - `qi::symbols` best
  - Function good
  - Lambda worse
  - Multiple lambdas in alternation worst
- All are O(1) with varying constants in this case
- `qi::symbols`'s complexity worsens when matching longer input or more elements
- Phoenix function allows calling functions without bind

Capturing the Specifier: Comparison

Writing to a UDT

- What if the `conversion_specifier` is in a struct?

```
struct conversion_specification
{
   conversion_specifier specifier;
};
```

- Use phoenix::bind
- Use a Phoenix function to write to the data member
- Adapt the struct for Fusion

Capturing the Specifier: phoenix::bind

```
struct conversion_specification
{
   conversion_specifier specifier;
};

qi::rule<It,conversion_specification()> specifier;
specifier
   = lit('%')
   >> char_("cdEeFfGginopsuXx")
      [
         phx::bind(&conversion_specification::specifier, _1)
      ]
   ;
```

Capturing the Specifier: Phoenix Function

```
specifier
   = lit('%')
   >> char_("cdEeFfGginopsuXx")
      [set_specifier(_val, _1)]
   ;
```

```
struct set_specifier_impl
{
   template <class, class>
   struct result { typedef void type; };
   template <class Val, class Char>
   void
   operator ()(Val & _val, Char _specifier) const
   {
      switch (_specifier)
      {
         char 'd': _val.specifier = CS_DECIMAL; break;
         char 'f': _val.specifier = CS_FIXED;   break;
         char 'g': _val.specifier = CS_GENERAL; break;
         ...
      }
   }
};
```

```
"%d"
"%f"
"%g"
```

set_specifier_impl

Capturing the Specifier: Adapted Struct

```
struct conversion_specification
{
   conversion_specifier specifier;
};

BOOST_FUSION_ADAPT_STRUCT(
   conversion_specification,
   (conversion_specifier, specifier)
)

qi::rule<It,conversion_specification()> specifier;
specifier
   = lit('%')
   >> char_("cdEeFfGginopsuXx")
   ;
```

Writing to a UDT

- What if the value must be set via a member function?

```
struct conversion_specification
{
   void set_specifier(char);
};
```

- Use boost::bind or phoenix::bind
- Use a Phoenix function to call the member function

set_specifier()

```
struct conversion_specification
{
    void
    set_specifier(char _specifier)
    {
        switch (_specifier)
        {
            char 'd': specifier = CS_DECIMAL; break;
            char 'f': specifier = CS_FIXED;   break;
            char 'g': specifier = CS_GENERAL; break;
            ...
        }
    }
};
```

"%d"
"%f"
"%g"

```
qi::rule<It,conversion_specification()> specifier;
specifier
   = lit('%')
   >> char_("cdEeFfGginopsuXx")
      [
         phx::bind(
            &conversion_specification::set_specifier,
            _val, _1)
      ]
   ;
```

Capturing the Specifier: phoenix::bind

Capturing the Specifier: Phoenix Function

```
specifier
   = lit('%')
   >> char_("cdEeFfGginopsuXx")
      [set_specifier(_val, _1)]
   ;
```

set_specifier_impl

```
struct set_specifier_impl
{
   ...
   template <class Val, class Char>
   void
   operator ()(Val & _val, Char _specifier) const
   {
      _val.set_specifier(_specifier);
   }
};
```

- Conversion specifications
  - %
  - **Flag characters (zero or more)**
  - Field width (optional)
  - Precision (optional)
  - Length modifier (optional)
  - Conversion specifier

Next: Flag Characters

Flag Characters

- - means left align
- # means alternate output
- 0 means fill with zeroes
- + means show the sign character
- <space> means insert a space for positive numbers
- ' means delimit thousands

```
"%-d"
"%#f"
"%0+g"
```

```
flags = char_("-#0+ '");
specification = lit('%') >> *flags >> specifier;
```

- Save as before

- Conversion specifications
  - %
  - Flag characters (zero or more)
  - **Field width (optional)**
  - Precision (optional)
  - Length modifier (optional)
  - Conversion specifier

Next: Flag Characters

Field Width

- An unsigned whole number

```
qi::rule<It,unsigned()> width;
width = qi::uint_;
specification
   = lit('%')
   >> *flag
   >> -width
   >> specifier
   ;
```

- Save as before

```
"%4d"
"%10f"
"%7g"
```

- Conversion specifications
  - %
  - Flag characters (zero or more)
  - Field width (optional)
  - **Precision (optional)**
  - Length modifier (optional)
  - Conversion specifier

Next: Precision

- Decimal point followed by an optional integer

```
qi::rule<It,int()> precision;
precision = lit('.') >> -qi::int_;
specification
   = lit('%')
   >> -flags
   >> -width
   >> -precision
   >> specifier
   ;
```

```
"%.d"
"%.6f"
"%.-1g"
```

- If there is no number after the decimal point, precision is zero
- If the number is negative, precision is zero

Precision

Precision: No Number

- If there is no number after the decimal point, precision is zero

```
qi::rule<It,int()> precision;
precision
    %= lit('.')[_val = val(0)]
    >> -qi::int_
    ;
```

```
"%.d"
```

Precision: No Number

- If there is no number after the decimal point, precision is zero

```
qi::rule<It,int()> precision;
precision
   = lit('.')
   >>
      (
         qi::int_
         | qi::attr(val(0))
      )
   ;
```

"%.d"

Precision: Negative Means Zero

- If the number is negative, precision is zero

```
qi::rule<It,int()> precision;
precision
   %= lit('.')
   >>
      (
         qi::int_
         [
            _val = _1,
            phx::if_(_1 < val(0))
            [
               _val = val(0)
            ]
         ]
         | qi::attr(val(0))
      )
   ;
```

"%.-1g"

Precision: Negative Means Zero

- If the number is negative, precision is zero

```
qi::rule<It,int()> precision;
precision
   = lit('.')
   >>
      (
         qi::int_
         [
            _val = _1,
            phx::if_(_1 < val(0))
```

Commas form Phoenix sequences, which require:

```
#include <boost/spirit/home/phoenix/statement/sequence.hpp>
```

```
         ]
         | qi::attr(val(0))
      )
   ;
```

Precision: Other Arguments

- `*n$` means nth argument supplies precision
- `*` means next argument supplies precision

```
qi::rule<It,int()> precision;
precision
   %= lit('.')
   >>
      (
         (lit('*') >> qi::int_ >> '$')[???]
         | lit('*')[???]
         | qi::int_
            [
               _val = _1,
               if_(_1 < val(0))[_val = val(0)]
            ]
         | qi::attr(val(0))
      )
   ;
```

"%*2$f"
"%*g"

- Conversion specifications
  - %
  - Flag characters (zero or more)
  - Field width (optional)
  - Precision (optional)
  - **Length modifier (optional)**
  - Conversion specifier

Next: Length Modifier

Length Modifier

- The length modifier can be one of the following:
  - h – short, unsigned short, or short * (with %n)
  - hh – signed char, unsigned char, or signed char * (with %n)
  - l – long, unsigned long, or long * (with %n), etc.
  - ll – long long, unsigned long long, or long long * (with %n)
  - L – long double
  - t – ptrdiff_t
  - z – size_t or ssize_t
  - A few others
- Not all supported on all platforms

```
"%hd"
"%Lg"
"%ln"
```

Microsoft adds their own:
I32 and I64

Length Modifier

```
modifier
   = lit("hh")
   | lit("ll")
   | char_("hlLtz")
   ;
```

```
"%hhd"
"%Lg"
"%zu"
```

- Need enumerated type for values
- Need semantic actions to save correct value

Length Modifier

```
enum length_modifier
{
   LM_SHORT,
   LM_CHAR,
   LM_LONG,
   LM_LONG_DOUBLE,
   LM_LONG_LONG,
   LM_SIZE_T,
   LM_PTRDIFF_T
};

modifier
   = lit("hh")     [_val = val(LM_SHORT)]
   | lit("ll")     [_val = val(LM_LONG_LONG)]
   | char_("hlLtz")[_val = get_length_modifier(_1)]
   ;
```

Specification Parser

```
specification
   = lit('%')
   >> *flags
   >> -width
   >> -precision
   >> -modifier
   >> specifier
   ;
```

```
"%#03hd"
"%-3.10Lf"
"%5.8Lg"
```

Parsing the Rest

- Can parse a conversion specification
- Must also parse
  - %%
  - Ordinary characters

```
"Rate: %1.3g%%"
```

```
format
   = *
      (
         (
            lit('%')
            >>
               (
                  '%'
                  | specification
               )
         )
         | char_
      )
   ;
```

The Full Format Parser

```
format
   = *
      (
         (
            lit('%')
            >>
               (
                  '%'
                  | specification
               )
         )
         | char_
      )
   ;
```

"Rate: "

The Full Format Parser

```
format
   = *
      (
         (
            lit('%')
            >>
               (
                  '%'
                  | specification
               )
         )
         | char_
      )
   ;
```

"Rate: %"

The Full Format Parser

The Full Format Parser

```
format
   = *
      (
         (
            lit('%')
            >>
               (
                  '%'
                  | specification
               )
         )
         | char_
      )
   ;
```

"Rate: %%"
"Rate: %1.3Lf%%"

```
specification
   = *flags
   >> -width
   >> -precision
   >> -modifier
   >> specifier
   ;
```

lit('%') is now part of the format rule

Fixing specification

The Full Format Parser

```
format
   = *
      (
         (
            lit('%')
            >>
               (
                  '%'
                  | specification
               )
         )
         | char_
      )
   ;
```

%% becomes % in the output

The Full Format Parser

```
format
   = *
      (
         (
            lit('%')
            >>
               (
                  '%'              [write(ref('%'))]
                  | specification
               )
         )
         | char_
      )
   ;
```

```
format
   = *
      (
         (
            lit('%')
            >>
               (
                  '%'              [write(ref('%'))]
                  | specification
               )
         )
         | char_
      )
   ;
```

Parses one specification. Could save offset in the output string and the specification data for post processing of the remaining arguments.

The Full Format Parser

The Full Format Parser

```
format
   = *
      (
         (
            lit('%')
            >>
               (
                  '%'             [write(ref('%'))]
                  | specification [save_or_convert(_1)]
               )
         )
         | char_
      )
   ;
```

```
format
   = *
      (
         (
            lit('%')
            >>
               (
                  '%'             [write(ref('%'))]
                  | specification [save_or_convert(_1)]
               )
         )
         | char_
      )
   ;
```

Some other character to copy to the output

The Full Format Parser

The Full Format Parser

```
format
   = *
      (
         (
            lit('%')
            >>
               (
                  '%'                [write(ref('%'))]
                  | specification [save_or_convert(_1)]
               )
         )
         | char_                        [write(_1)]
      )
   ;
```

Debugging the Parser

- qi::debug(rule);
- rule.name("rule");

Must precede debug(rule) to appear in debugging output!

Debugging the Parser

- qi::debug(rule);
- rule.name("rule");
- BOOST_SPIRIT_DEBUG_NODE(rule);
  - Does both (in correct order!)
  - May need names when not debugging
- BOOST_SPIRIT_DEBUG
- http://boost-spirit.com/home/articles/doc-addendum/debugging/

Configuring for Debugging

```
rule1 = ...;
rule1.name("rule1");
rule2 = ...;
rule2.name("rule2");
...

#ifdef BOOST_SPIRIT_DEBUG
debug(rule1);
debug(rule2);
...
#endif
```

```
rule1 = ...;

rule2 = ...;
...



BOOST_SPIRIT_DEBUG_NODE(rule1);
BOOST_SPIRIT_DEBUG_NODE(rule2);
...
```

Debug Output Without Names

```
...
<unnamed-rule>
  <try>s has %s a %s</try>
  <unnamed-rule>
    <try>s has %s a %s</try>
    <fail/>
  </unnamed-rule>
  <unnamed-rule>
    <try>s has %s a %s</try>
    <fail/>
  </unnamed-rule>
  <unnamed-rule>
    <try>s has %s a %s</try>
    <fail/>
  </unnamed-rule>
  <unnamed-rule>
    ...
```

Debug Output With Names

```
...
<specification>
  <try>s has %s a %s</try>
  <flags>
    <try>s has %s a %s</try>
    <fail/>
  </flags>
  <width>
    <try>s has %s a %s</try>
    <fail/>
  </width>
  <precision>
    <try>s has %s a %s</try>
    <fail/>
  </precision>
  <modifier>
    ...
```

Expectation Points

- Sometimes a grammar requires what follows
- Failure to match is an error
- Consider a grammar expecting an IP address next

```
uint_ >> '.' >> uint_ >> '.' >> uint_ >> '.' >> uint_
```

- That parser can consume some input before failing
- Use expectation points

```
uint_ > '.' > uint_ > '.' > uint_ > '.' > uint_
```

Expectation Points

- *Expectation point* operator: >
- Used instead of *follows* (>>) operator
- Right hand parser *must* match, since previous parsers matched
- Failure triggers `qi::expectation_failure` exception
- Add `qi::on_error` handler to report
  - Name of the parser that failed
  - Name of rule if rule is on the right hand side

Adding an Expectation Point

```
format
   = *
      (
         (
            lit('%')
            >
               (
                  '%'
                  | specification
               )
         )
         | char_
      )
   ;
```

```
format
   = *
      (
         (
            lit('%')
            >
               (
                  '%'
                  | specification
               )
         )
         | char_
      )
   ;
qi::on_error<fail>(format, handler);
```

Reporting Errors: on_error

Reporting Errors: on_error

```
on_error<fail>(format, handler);
```

- Four documented values available to the error handler
  - The input range: [_1, _2)
  - The position where the error was detected: _3
  - The *what* string: _4
- Other values are available:
  - Synthesized attribute: _val
  - Local variables: _a, _b, ...
  - Inherited attributes: _r1, _r2, ...

```
format = *((lit('%') > ('%' | specification)) | char_);
on_error<fail>(format,
   std::cerr << ref("Expected ") << _4 << std::endl
);
```

Reporting Errors

Reporting Errors

```
format = *((lit('%') > ('%' | specification)) | char_);
on_error<fail>(format,
   std::cerr << ref("Expected ") << _4 << std::endl
);
```

In this case _4 will be:

`<alternative>"%"<specification>`

Reporting Errors

```
format = *((lit('%') > ('%' | specification)) | char_);
on_error<fail>(format,
   std::cerr << ref("Expected ") << _4 << std::endl
);
```

In this case _4 will be:

```
<alternative>"%"<specification>
```

```
format = *((lit('%') > ('%' | specification)) | char_);
on_error<fail>(format,
   std::cerr << ref("Expected ") << _4 << std::endl
);
```

In this case _4 will be:

```
<alternative>"%"<specification>
```

Reporting Errors

```
format = *((lit('%') > ('%' | specification)) | char_);
on_error<fail>(format,
   std::cerr << ref("Expected ") << _4 << std::endl
);
```

In this case _4 will be:

```
<alternative>"%"<specification>
```

Reporting Errors

Reporting Clearer Errors

```
format = *((lit('%') > ('%' | specification)) | char_);
on_error<fail>(format,
   ref(std::cerr)
      << "Expected another '%' or a specification "
         "after '%'"
      << std::endl
);
```

Reporting Error Context

- Reporting an error without context is unhelpful
- `[_1, _2)` is input available to the failing parser
- `_3` refers to a position within `[_1, _2)`
- Can produce a message that reports `[_1, _2)` and points to `_3` within
- Newlines complicate the logic to get pretty output

Wrapping Up

- Determine approach to use:
  - Parse entire format string and convert later
  - Convert while parsing
- Determine how to store specification state
- Add conversion logic
- Add error handling to report misuse
- Extend the format with
  - New conversion specifiers for UDTs
  - New format flags

Next Steps

- Documentation (latest http://www.boost.org/libs/spirit)
- http://boost-spirit.com
- Spirit-general mailing list
- #boost IRC channel
- E-mail: robert.stewart@sig.com

More Information