GHF automation GmbH

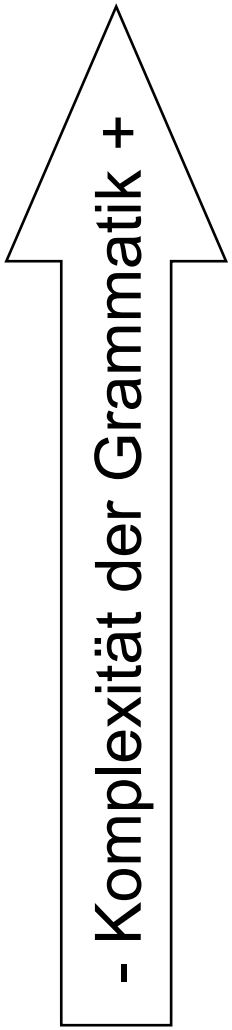
Einführung in Boost Spirit Qi

Das C++ Parser Framework

© 2014 Georg Hellack

Warum noch ein Parser?

GHF automation GmbH



Lex/Yacc, AntLR

Boost Spirit

istringstream,
lexical_cast

+ Grammatik im Quellcode

+ kein externes Tool

+ UNICODE Support

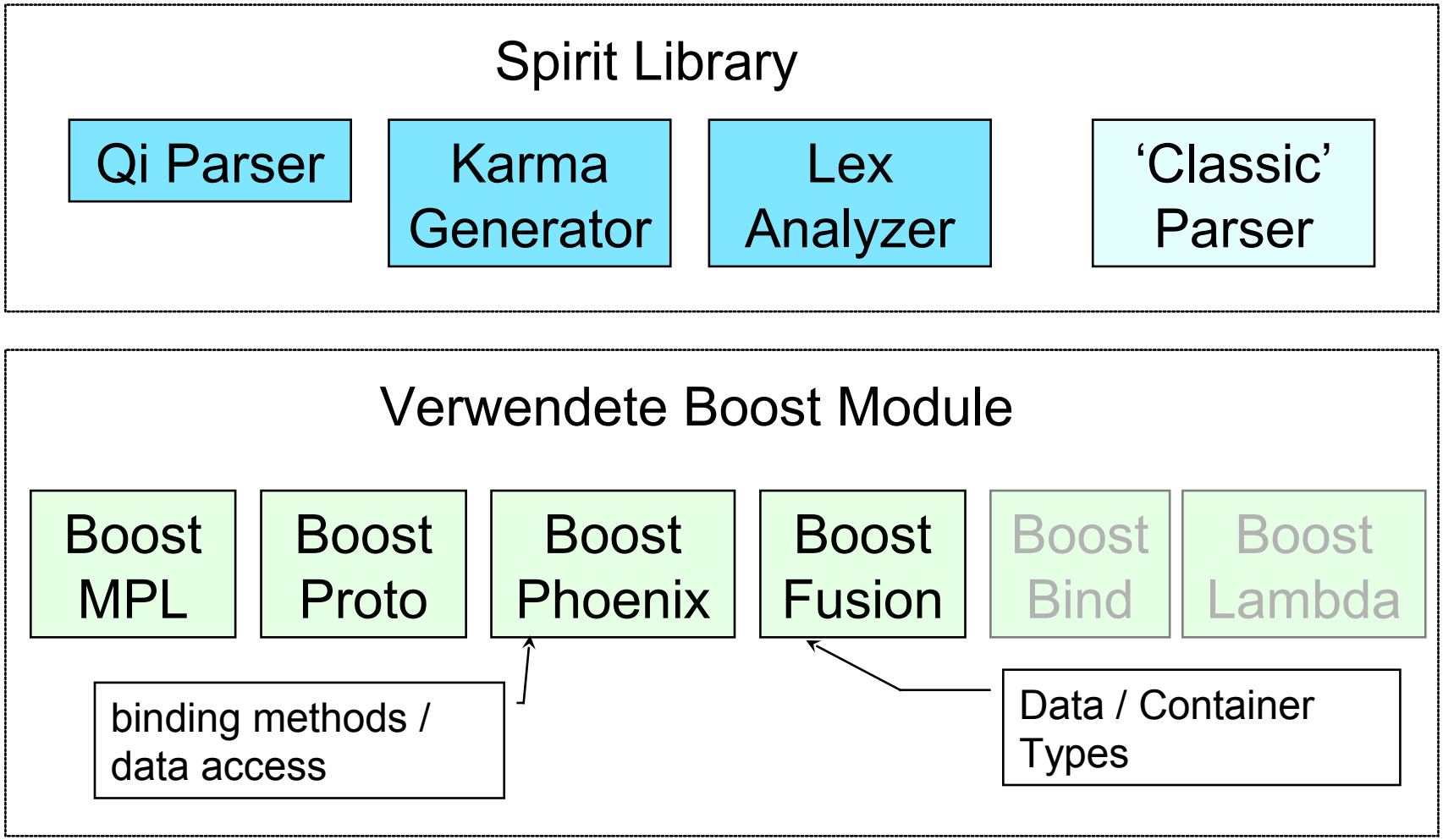
- lange Compilezeiten

- verwirrende Vielfalt
(“viele Wege führen auch nach Rom”)

© 2014 Georg Hellack

Spirit im Boost Kontext

GHF automation GmbH



© 2014 Georg Hellack

Es ginge auch ohne Spirit…

GHF automation GmbH

```
std::istringstream l_is("123");
l_is >> l_iVal;
```

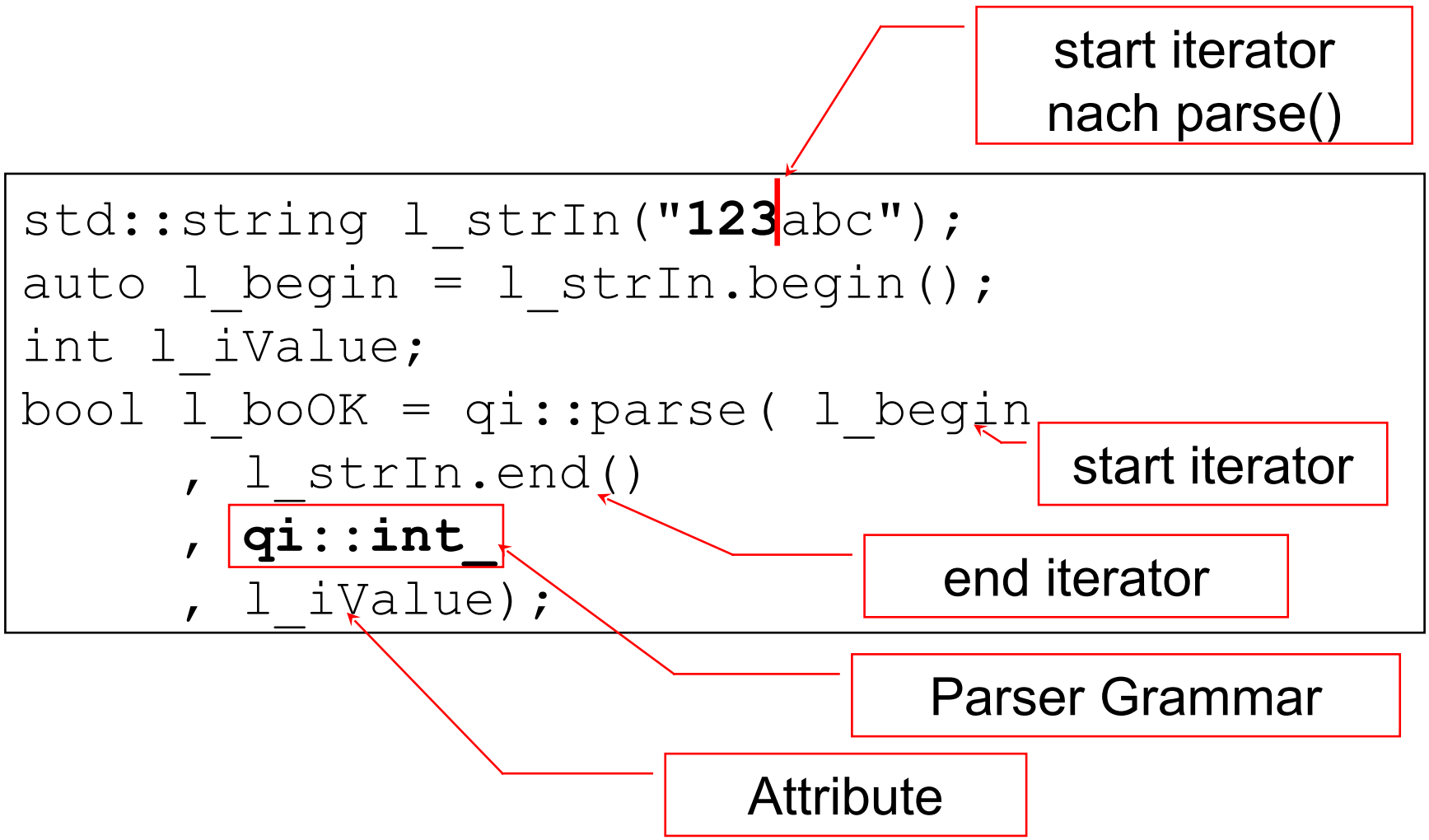
```
try{  l_iVal=boost::lexical_cast<int>("
123");
}catch(const boost::bad_lexical_cast &)
{}
```

```
std::string l_strIn("123");
int l_iValue;
qi::parse(l_strIn.begin(),l_strIn.end()
          , qi::int_
          , l_iValue);
```

© 2014 Georg Hellack

Parser Aufruf – qi::parse()

GHF automation GmbH



© 2014 Georg Hellack

Parser Aufruf – qi::phrase_parse()

GHF automation GmbH

qi::phrase_parse erlaubt Angabe eines weiteren Parsers, um Teile des Inputs zu überspringen, z.B. ‘Whitespace’ oder auch Kommentare

```
std::string l_strIn("123abc");
auto l_begin = l_strIn.begin();
int l_iValue;
bool l_boOK = qi::phrase_parse( l_begin
     , l_strIn.end()
     , qi::int_
     , qi::space
     , l_iValue);
```

Standard Wsp-Parser als Skipper

© 2014 Georg Hellack

Parser Organisieren – Rule

GHF automation GmbH

qi::rule verbindet eine Parser-Grammatik mit 'Return-Wert' (Attribute), Parser-lokalen Variablen und Debug-Ausgabe

```
std::string l_strInput( "123+456" );
...
qi::rule<std::string::iterator, int()
 , qi::space_type, qi::locals<int>> l_rule
= qi::int_[qi::_a = qi::_1]
  >> qi::lit('+')
  >> qi::int_[qi::_val = qi::_a + qi::_1];
```

Skipper Typ

Iterator Typ

Rule-Attribute

Attribut int_-Parser

Result-Attribut, d.h. das Attribut der Rule

rule-lokale Variable

© 2014 Georg Hellack

Parser Organisieren – Grammar

GHF automation GmbH

qi::grammar verknüpft verschiedene Rules

```
template <typename Iterator>
class CMyGrammar : public qi::grammar<Iterator, std::vector<int>(),
qi::space_type> {
public:
  CMyGrammar () : CMyGrammar::base_type( m_start ) {
    m_start = m_rule1 >> -m_rule2;
    m_rule1 = qi::int_;
    m_rule2 = '#' >> qi::int_;
  }
  qi::rule<Iterator, std::vector<int>(), qi::space_type> m_start;
  qi::rule<Iterator, int(), qi::space_type> m_rule1;
  qi::rule<Iterator, int(), qi::space_type> m_rule2;
};
...
std::string l_strInp("4711#0815");
std::vector<int> l_aiData;
CMyGrammar<std::string::iterator> l_grammar;
qi::phrase_parse( l_strInp.begin(), l_strInp.end(), l_grammar,
qi::space_type(), l_aiData );
```

2. rule ist optional (‘-’)

grammar Attribut

Puffer für Attribut

grammar Objekt übergeben

© 2014 Georg Hellack

Basic Parsers

GHF automation GmbH

- **Character** Parser, e.g. char_, alnum, alpha, space
- **Numerische** Parser, e.g. int_, float_, hex
- **String** Parser, e.g. lit(“string”), symbols
- **Hilfs**parser, e.g. eol, eoi, lazy(), attr()
- **Binäre** Parser, e.g. byte_, little_word, big_dword

© 2014 Georg Hellack

Directives

GHF automation GmbH

- **lexeme**[*parser*]: ‘skipping’ im Parser ausschalten, z.B. “1 23 456” ist nicht “123456”
- **omit**[*parser*]: ‘verschluckt’ das Attribut des Parsers
- **as_string**[*parser*]: konvertiert Attribut in std::string
- **matches**[*parser*]: liefert ein bool-Attribut, ob Parser erfolgreich war
- **repeat**(min, max)[*parser*]: steuert die Anzahl Aufrufe des wiederholenden Parsers
- **raw**[*parser*]: liefert Begin- und End-Iterator des vom Parser ‘gematchten’ Bereichs

© 2014 Georg Hellack

Operators

GHF automation GmbH

parser	0 oder mehrere
+parser	1 oder mehrere
-parser	0 oder einmal, i.e. optional
!parser	‘not’ predicate
&parser	‘and’ predicate
parserA **|** *parserB*	Alternative (A oder B)
parserA **>>** *parserB*	Sequenz (B folgt A)
parserA **>** *parserB*	Expectation (B muß auf A folgen)
parserA **-** *parserB*	Differenz (A, aber nicht B)
parserA **||** *parserB*	Oder-Sequenz (A oder B oder A gefolgt von B)
parserA **%** *parserB*	Liste (Folge von As, getrennt durch Bs)
parserA **^** *parserB*	Permutation (wie ‘||’, aber Reihenfolge egal)

© 2014 Georg Hellack

Beispiele

GHF automation GmbH

1. Keyword ‘=‘ Value:

```
+(char_ - '=') >> '=' >> int_
```

2. Liste von Integern

```
int_ % ',' oder int_ >> *(',' >> int_)
```

3. Buchstabe oder Ziffer:

```
alpha | digit
```

4. Keyword : Integer (optional, default 0)

```
+(char_-':') >> ':' >> (int_|attr(0))
```

5. Integer gefolgt von einem ‘#’

```
int_ >> &char_('#')
```

© 2014 Georg Hellack

Parser Attribute - Propagation

GHF automation GmbH

- Parser Attribute sind die ‘Ergebnis-Werte’ eines Parsers; sie sind im Spirit Manual gelistet, z.B.:

Expression	Attribute	Description
ch	Unused	Matches ch
lit(ch)	Unused	Matches ch
char_	Ch	Matches any character
char_(ch)	Ch	Matches ch

-Attribute einzelner Parser werden zu einem Fusion Vector zusammengesetzt

```
fu::vector<std::string, int
     , boost::optional<std::string>> l_aData;
…
l_rule = +(qi::alpha) >> ":" >> qi::int_
     >> -(':' >> +(qi::alpha));
```

© 2014 Georg Hellack

Parser Attribute - Compatibility

GHF automation GmbH

- Attribute-Typen werden automatisch umgesetzt, z.B.
  - *char_ liefert den Attribut-Typ ‘std::vector<char>’, kann aber in std::string zugewiesen werden
- mittels der Hilfsklasse ‘’ kann der Attribut-Typ eines Parser-Ausdrucks angezeigt werden:

```
display_attribute_of_parser(
      *(qi::alpha) >> -(':' >> qi::int_)
);
```

liefert:

```
struct boost::fusion::vector2<
  class std::vector<char,class std::allocator<char>>
  ,class boost::optional<int> >
```

© 2014 Georg Hellack

Anzeige des Attribut-Typs

GHF automation GmbH

Template Klassen von Hartmut Kaiser

(http://boost-spirit.com/home/2010/01/31/what-is-the-attribute-type-exposed-by-a-parser/)

```
template <typename Expr, typename Iterator =
boost::spirit::unused_type>
struct attribute_of_parser {
  typedef typename boost::spirit::result_of::compile<qi::domain
        , Expr>::type parser_expression_type;
  typedef typename boost::spirit::traits::attribute_of<
      parser_expression_type, boost::spirit::unused_type, Iterator
  >::type type;
};

template <typename T>
void display_attribute_of_parser (T const&)
{
  typedef typename attribute_of_parser<T>::type attribute_type;
  std::cout << typeid(attribute_type).name() << std::endl;
}
```

© 2014 Georg Hellack

Semantic Actions

GHF automation GmbH

- dem Parser mit ‘[]’ nachgestellt
- werden vom jeweiligen Parser bei Erfolg ausgelöst
- erhalten den Parser-Kontext mit
    - qi::_1, qi::_2 ...: Attribut des zugehörigen Parsers
    - qi::_val: Ergebnis-Attribut des übergeordneten Parsers
    - qi::_r1, …: geerbte Attribute des übergeordneten Parsers
    - qi::_a, …: lokale Variablen
    - qi::_pass: erlaubt Abbruch des Parsers
- zweitrangig gegenüber Attribute-Propagation

© 2014 Georg Hellack

Semantic Actions – Call Method

GHF automation GmbH

```
struct t_Data {
  void SetCity(const std::string& p_szCity);
  void SetCode(int p_iAreaCode);
};
…
qi::rule<std::string::iterator, t_Data()>
l_rule = boost::spirit::as_string[+qi::alnum]
[px::bind(&t_Data::SetCity,&qi::_val,qi::_1)]
 >> ':' >> qi::int_
[px::bind(&t_Data::SetCode,&qi::_val,qi::_1)];
```

Ziel-Attribut ist eine Klasse

Achtung: keine Attribute-Propagation bei Semantic Actions; daher `as_string[std::vector<char>]` -> `std::string`!

Check Define **`BOOST_SPIRIT_ACTIONS_ALLOW_ATTR_COMPAT`** !

© 2014 Georg Hellack

Noch mehr Semantic Actions

GHF automation GmbH

Zuweisung zu Daten-Membern

```
struct t_Data {
  int m_iCode;
};
…
[px::bind(&t_Data::m_iCode,qi::_val)= qi::_1]
```

Zuweisung zu lokalen Variablen

```
std::string l_strCity;
…
[px::ref(l_strCity) = qi::_1]
```

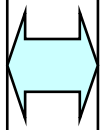
© 2014 Georg Hellack

Fusion Vectors

GHF automation GmbH

- Container für Elemente unterschiedlichen Typs in fester Reihenfolge
- ‘optional’-Typ mit Prüfung, ob Wert gesetzt, für optionale Parser-Attribute (e.g. –qi::int_)
- Zugriff auf Elemente: **fu::at_c<***index***>(***varFusionVector***)**
- C++-struct als Vector: ‘BOOST_FUSION_ADAPT_STRUCT’

```
struct t_Data
{
  std::string m_strCity;
  int m_iAreaCode;
};
```



```
BOOST_FUSION_ADAPT_STRUCT(
  t_Data,
  (std::string, m_strCity)
  (int, m_iAreaCode)
)
```

© 2014 Georg Hellack

Fehlersuche mit Debug Output

GHF automation GmbH

Einfügen von Debug-Info in qi::rule() durch

- Benennung: `myRule.name("myRule")`
- Debug Info: `qi::debug(myRule)`
- oder einfach: `BOOST_SPIRIT_DEBUG_NODE(myRule)`

erzeugt einen XML-Dump des Parsers

```
<l_rule>
  <try>Aachen:52062</try>
  <success></success>
  <attributes>[[[A, a, c, h, e, n], 52062]]</attributes>
</l_rule>
```

```
<l_rule>
  <try>Aachen:ABC</try>
  <fail/>
</l_rule>
```

© 2014 Georg Hellack

Fehlersuche mit Exceptions

GHF automation GmbH

- Parser Exceptions können ausgelöst werden mittels ‘Expectation’ Operator (‘**>**’) oder ‘Expectation Point’ (qi::eps)
- Exception liefert, wo der Fehler auftrat ([qi::_2, qi::_3]) und was erwartet wurde (qi::_4)

Beispiel: für Input “Aachen:ABC” liefert der Parser unten

```
Error! Expecting <sequence>":"<integer> here: ":ABC"
Unparsed input 'Aachen:ABC'
```

```
qi::rule<std::string::iterator, t_Data()> l_rule =
*qi::alnum > (':' >> qi::int_); qi::on_error<qi::fail>(
l_rule,
    std::cout << px::val("Error! Expecting ")
      << qi::_4 << " here: \""
      << px::construct<std::string>( qi::_3, qi::_2 )
      << "\"\n") );
```

© 2014 Georg Hellack

Der Symbols–Parser als Map

GHF automation GmbH

Der **qi::symbols** Parser arbeitet als map; der eingelesene Key liefert den zugeordneten Value als Attribute.

```
qi::symbols<char, int> l_symPLZ;
l_symPLZ.add( "Aachen", 52062 );
l_symPLZ.add( "Bonn", 53111 );
l_symPLZ.add( "Hamburg", 20095 );

std::string l_strInp("Bonn");
int l_iPLZ;
qi::parse( l_strInp.begin()
  , l_strInp.end(), l_symPLZ, l_iPLZ );
```

Ergebnis ist: `53111`

© 2014 Georg Hellack

Symbols–Parser als Parser-Control

GHF automation GmbH

Die Values im **qi::symbols** Parser können auch Pointer auf qi::rule Objekte sein. Dadurch kann ein Keyword den weiteren Parser steuern (sogenannter Nabialek-Trick).

```
typedef
qi::rule<Iterator,t_SymValue(),Skipper> t_SymRuleType;
qi::symbols<char, t_SymRuleType*> m_ruleSymbols;
t_SymRuleType m_rAddrHex, m_rAddrDec;
…
m_ruleSymbols.add( "hex", &m_rAddrHex );
m_ruleSymbols.add( "dec", &m_rAddrDec );
…
m_rDevice %= qi::omit[m_ruleSymbols[qi::_a = qi::_1]]
      >> qi::lazy(*qi::_a);
```

rule-Pointer in lokale Variable

Aufruf der rule

© 2014 Georg Hellack

Binary Parser

GHF automation GmbH

- vorh. Parser für Bytes, Integer in Little- und Big –Endian
- **qi:advance**(nr bytes) Parser, um Bytes zu überspringen

Beispiel: Puffer enthält ein 2Byte-Feld mit der Länge des folgenden Bereichs, danach weitere Daten
0x4,0x0,*0x1,0x2,0x3,0x4*,0x5,0x6

```
std::vector<unsigned char> l_abData;
auto l_posStart = l_abData.begin();

qi::rule<std::vector<unsigned char>::iterator
   , qi::locals<int>> l_rule
qi::little_word[qi::_a = qi::_1]
>> repository::qi::advance(qi::_a);

qi::parse( l_posStart, l_abData.end(), l_rule );
```

Länge in Variable speichern

parsed Länge

Skip Länge

© 2014 Georg Hellack

Caveats I

GHF automation GmbH

- Defines vor dem Einbinden von boost/spirit/include/qi.hpp setzen, i.e. `BOOST_SPIRIT_USE_PHOENIX_V3` und ggf. `BOOST_SPIRIT_DEBUG` und `BOOST_SPIRIT_ACTIONS_ALLOW_ATTR_COMPAT`
- Namespaces für Platzhalter bei Phoenix Bind (qi::_1…) und Boost Bind (::_1) beachten (Phoenix bevorzugen)
- qi::char_ liest Trennzeichen -> Differenz-Operator, e.g. `*(char-':')`
- Klassen in spirit/repository beachten, z.B. **distinct**, **confix**, **kwd**
- bei einer **rule** nicht Skipper-Parameter vergessen
- 'Semantic Actions' unterbinden Attribute Propagation, daher Datentyp beachten

© 2014 Georg Hellack

Caveats II

GHF automation GmbH

- Klammern beim Attribut-Typ: `qi::rule<it, t_Data()> l_rule`

```
std::string l_strInput( "123+456" );
int l_iSum;
qi::rule<std::string::iterator, int(),
qi::locals<int>> l_rule = qi::int_[qi::_a=qi::_1] >>
'+' >> qi::int_[qi::_val = qi::_a + qi::_1];
```

ohne ()

```
Running 1 test case...
<l_rule>
  <try>123 + 456</try>
  <success></success>
  <attributes>[]</attributes><locals>(123)</locals>
</l_rule>
parse completed; result is -858993460
```

mit ()

```
Running 1 test case...
<l_rule>
  <try>123 + 456</try>
  <success></success>
  <attributes>[579]</attributes><locals>(123)</locals>
</l_rule>
parse completed; result is 579
```

© 2014 Georg Hellack

Die Zukunft

GHF automation GmbH

Release-Version ist 2.5.3 (Boost Version 1.50.0)

Die kommende Version ist ‘Spirit X3’. Sie wird vorallem C++11 Features nutzen.

- Customization Points
- Support von C++11 Lambda Funktionen in ‘Semantic Actions’
- kürzere Compile-Zeiten

© 2014 Georg Hellack

Referenzen und Links

GHF automation GmbH

- Guzman, J, und Kaiser, H.: Spirit 2.5 Manual (http://boost-spirit.com/dl_docs/spirit2_5.pdf)
- Caisse, M.: Using Spirit 2.3, BoostCon 2010 (http://boost-spirit.com/home/wp-content/uploads/2010/05/spirit_presentation.pdf)
- Thomson, R.: Boost Spirit 2.5.2 Reference Card (http://user.xmission.com/~legalize/spirit/spirit-reference.pdf)
- Guzman, J.: “Inside Spirit X3 - Redesigning Boost.Spirit for C++11“, Vortrag während ‚C++ Now!‘, 2013 (http://www.youtube.com)
- Patisserie Lints, Antwerpen (http://lints.be/)

© 2014 Georg Hellack

Und darauf ein…

GHF automation GmbH

© 2014 Georg Hellack