Table of Contents

Preface ................................................................................................................................. 4
What's New .......................................................................................................................... 7
   Spirit V2.5 .................................................................................................................. 7
   Spirit V2.4.2 .......................................................................................................... 9
   Spirit V2.4.1 .......................................................................................................... 10
   Spirit V2.4 .......................................................................................................... 11
   Spirit V2.3 .......................................................................................................... 12
   Spirit V2.2 .......................................................................................................... 13
   Spirit V2.1 .......................................................................................................... 14
   Spirit Classic ....................................................................................................... 16
Introduction ..................................................................................................................... 17
Structure ......................................................................................................................... 22
   Include .............................................................................................................. 22
Attributes ....................................................................................................................... 29
   Attributes of Primitive Components .................................................................. 29
   Attributes of Compound Components ............................................................... 30
   More About Attributes of Compound Components ............................................ 31
   Attributes of Rules and Grammars ....................................................................... 33
Qi - Writing Parsers ....................................................................................................... 34
Tutorials ......................................................................................................................... 34
   Quick Start ....................................................................................................... 34
   Warming up ....................................................................................................... 34
   Parser Semantic Actions .................................................................................. 37
   Complex - Our first complex parser ................................................................. 39
   Sum - adding numbers .................................................................................... 40
   Number List - stuffing numbers into a std::vector ........................................... 42
   Number List Redux - list syntax .................................................................. 43
   Number List Attribute - one more, with style ............................................... 44
   Roman Numerals ............................................................................................ 45
   Employee - Parsing into structs .................................................................. 50
   Mini XML - ASTs? ......................................................................................... 55
   Mini XML - Error Handling .......................................................................... 61
Quick Reference .......................................................................................................... 64
   Common Notation .......................................................................................... 64
   Qi Parsers ...................................................................................................... 66
   Compound Attribute Rules ........................................................................... 74
   Nonterminals ................................................................................................. 78
   Parser Semantic Actions .............................................................................. 79
   Phoenix .......................................................................................................... 80
Reference ....................................................................................................................... 80
This is the documentation of the newest version of Spirit (currently, V2.5). If you're looking for the documentation of Spirit's previous version (formerly Spirit V1.8), see Spirit Classic.
Preface

“Examples of designs that meet most of the criteria for "goodness" (easy to understand, flexible, efficient) are a recursive-descent parser, which is traditional procedural code. Another example is the STL, which is a generic library of containers and algorithms depending crucially on both traditional procedural code and on parametric polymorphism.” --Bjarne Stroustrup

History

80s

In the mid-80s, Joel wrote his first calculator in Pascal. Such an unforgettable coding experience, he was amazed at how a mutually recursive set of functions can model a grammar specification. In time, the skills he acquired from that academic experience became very practical as he was tasked to do some parsing. For instance, whenever he needed to perform any form of binary or text I/O, he tried to approach each task somewhat formally by writing a grammar using Pascal-like syntax diagrams and then a corresponding recursive-descent parser. This process worked very well.

90s

The arrival of the Internet and the World Wide Web magnified the need for parsing a thousand-fold. At one point Joel had to write an HTML parser for a Web browser project. Using the W3C formal specifications, he easily wrote a recursive-descent HTML parser. With the influence of the Internet, RFC specifications were abundant. SGML, HTML, XML, email addresses and even those seemingly trivial URLs were all formally specified using small EBNF-style grammar specifications. Joel had more parsing to do, and he wished for a tool similar to larger parser generators such as YACC and ANTLR, where a parser is built automatically from a grammar specification.

This ideal tool would be able to parse anything from email addresses and command lines, to XML and scripting languages. Scalability was a primary goal. The tool would be able to do this without incurring a heavy development load, which was not possible with the above mentioned parser generators. The result was Spirit.

Spirit was a personal project that was conceived when Joel was involved in R&D in Japan. Inspired by the GoF's composite and interpreter patterns, he realized that he can model a recursive-descent parser with hierarchical-object composition of primitives (terminals) and composites (productions). The original version was implemented with run-time polymorphic classes. A parser was generated at run time by feeding in production rule strings such as:

```
prod ::= {'A' | 'B'} 'C';
```

A compile function compiled the parser, dynamically creating a hierarchy of objects and linking semantic actions on the fly. A very early text can be found here: pre-Spirit.

2001 to 2006

Version 1.0 to 1.8 was a complete rewrite of the original Spirit parser using expression templates and static polymorphism, inspired by the works of Todd Veldhuizen (Expression Templates, C++ Report, June 1995). Initially, the static-Spirit version was meant only to replace the core of the original dynamic-Spirit. Dynamic-Spirit needed a parser to implement itself anyway. The original employed a hand-coded recursive-descent parser to parse the input grammar specification strings. It was at this time when Hartmut Kaiser joined the Spirit development.

After its initial "open-source" debut in May 2001, static-Spirit became a success. At around November 2001, the Spirit website had an activity percentile of 98%, making it the number one parser tool at Source Forge at the time. Not bad for a niche project like a parser library. The "static" portion of Spirit was forgotten and static-Spirit simply became Spirit. The library soon evolved to acquire more dynamic features.

Spirit was formally accepted into Boost in October 2002. Boost is a peer-reviewed, open collaborative development effort around a collection of free Open Source C++ libraries covering a wide range of domains. The Boost Libraries have become widely known as an industry standard for design and implementation quality, robustness, and reusability.
Over the years, especially after Spirit was accepted into Boost, Spirit has served its purpose quite admirably. Classic-Spirit (versions prior to 2.0) focused on transduction parsing, where the input string is merely translated to an output string. Many parsers fall into the transduction type. When the time came to add attributes to the parser library, it was done in a rather ad-hoc manner, with the goal being 100% backward compatible with Classic Spirit. As a result, some parsers have attributes, some don't.

Spirit V2 is another major rewrite. Spirit V2 grammars are fully attributed (see Attribute Grammar) which means that all parser components have attributes. To do this efficiently and elegantly, we had to use a couple of infrastructure libraries. Some did not exist, some were quite new when Spirit debuted, and some needed work. Boost.Mpl is an important infrastructure library, yet is not sufficient to implement Spirit V2. Another library had to be written: Boost.Fusion. Fusion sits between MPL and STL -- between compile time and runtime -- mapping types to values. Fusion is a direct descendant of both MPL and Boost.Tuples. Fusion is now a full-fledged Boost library. Phoenix also had to be beefed up to support Spirit V2. The result is Boost.Phoenix. Last but not least, Spirit V2 uses an Expression Templates library called Boost.Proto.

Even though it has evolved and matured to become a multi-module library, Spirit is still used for micro-parsing tasks as well as scripting languages. Like C++, you only pay for features that you need. The power of Spirit comes from its modularity and extensibility. Instead of giving you a sledgehammer, it gives you the right ingredients to easily create a sledgehammer.

**New Ideas: Spirit V2**

Just before the development of Spirit V2 began, Hartmut came across the StringTemplate library that is a part of the ANTLR parser framework.¹ The concepts presented in that library lead Hartmut to the next step in the evolution of Spirit. Parsing and generation are tightly connected to a formal notation, or a grammar. The grammar describes both input and output, and therefore, a parser library should have a grammar driven output. This duality is expressed in Spirit by the parser library Spirit.Qi and the generator library Spirit.Karma using the same component infrastructure.

The idea of creating a lexer library well integrated with the Spirit parsers is not new. This has been discussed almost since Classic-Spirit (pre V2) initially debuted. Several attempts to integrate existing lexer libraries and frameworks with Spirit have been made and served as a proof of concept and usability (for example see Wave: The Boost C/C++ Preprocessor Library, and SLex: a fully dynamic C++ lexer implemented with Spirit). Based on these experiences we added Spirit.Lex: a fully integrated lexer library to the mix, allowing the user to take advantage of the power of regular expressions for token matching, removing pressure from the parser components, simplifying parser grammars. Again, Spirit's modular structure allowed us to reuse the same underlying component library as for the parser and generator libraries.

**How to use this manual**

Each major section (there are 3: Qi, Karma, and Lex) is roughly divided into 3 parts:

1. Tutorials: A step by step guide with heavily annotated code. These are meant to get the user acquainted with the library as quickly as possible. The objective is to build the confidence of the user in using the library through abundant examples and detailed instructions. Examples speak volumes and we have volumes of examples!

2. Abstracts: A high level summary of key topics. The objective is to give the user a high level view of the library, the key concepts, background and theories.

3. Reference: Detailed formal technical reference. We start with a quick reference -- an easy to use table that maps into the reference proper. The reference proper starts with C++ concepts followed by models of the concepts.

Some icons are used to mark certain topics indicative of their relevance. These icons precede some text to indicate:

---

¹ Quote from http://www.stringtemplate.org/: It is a Java template engine (with ports for C# and Python) for generating source code, web pages, emails, or any other formatted text output.
### Table 1. Icons

<table>
<thead>
<tr>
<th>Icon</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Note Icon" /></td>
<td>Note</td>
<td>Generally useful information (an aside that doesn't fit in the flow of the text)</td>
</tr>
<tr>
<td><img src="image" alt="Tip Icon" /></td>
<td>Tip</td>
<td>Suggestion on how to do something (especially something that is not obvious)</td>
</tr>
<tr>
<td><img src="image" alt="Important Icon" /></td>
<td>Important</td>
<td>Important note on something to take particular notice of</td>
</tr>
<tr>
<td><img src="image" alt="Caution Icon" /></td>
<td>Caution</td>
<td>Take special care with this - it may not be what you expect and may cause bad results</td>
</tr>
<tr>
<td><img src="image" alt="Danger Icon" /></td>
<td>Danger</td>
<td>This is likely to cause serious trouble if ignored</td>
</tr>
</tbody>
</table>

This documentation is automatically generated by Boost QuickBook documentation tool. QuickBook can be found in the Boost Tools.

### Support

Please direct all questions to Spirit's mailing list. You can subscribe to the [Spirit General List](mailto:spirit.general@lists.boost.org). The mailing list has a searchable archive. A search link to this archive is provided in Spirit's home page. You may also read and post messages to the mailing list through [Spirit General NNTP news portal](http://news.gmane.org/gmane.comp.parsers.spirit.general) (thanks to Gmane). The news group mirrors the mailing list. Here is a link to the archives: [http://news.gmane.org/gmane.comp.parsers.spirit.general](http://news.gmane.org/gmane.comp.parsers.spirit.general).
What's New

Spirit V2.5

What's changed in Spirit.Qi and Spirit.Karma from V2.4.2 (Boost V1.46.0) to V2.5 (Boost V1.47.0)

This version of Spirit now supports the new version of Boost.Phoenix (V3), which has been released as part of Boost V1.47. Please note that this support is purely preliminary and should not be considered production quality. The reason why we are releasing this now is to enable people who want to use the new version of Boost.Phoenix in their code to do so without any conflicts with existing code using Spirit. Generally, no Spirit code needs to be changed. To activate the use of Boost.Phoenix V3 for Spirit you need to define the following preprocessor constant for your builds (before including any of Spirit's include files):

```
#define BOOST_SPIRIT_USE_PHOENIX_V3 1
```

New Features in Qi or Karma

• Added `utree`, a generic, hierarchical, and dynamic data structure that can represent abstract syntax trees. It's well integrated with Spirit.Qi and Spirit.Karma. It can be passed as an attribute while parsing to almost any grammar. At the same time, it can be used as an attribute to generate output from.

• Added a new macro `BOOST_SPIRIT_TERMINAL_NAME` which takes in two parameters (the terminal name and its type). Before, there was only one parameter (the terminal name) and the type assumed the terminal name with `_type` appended. This produced invalid C++ identifiers with terminals such as `int_`, which resulted in generated a type `int__type` with a bogus double underscore that is reserved for system use in C++.

• The numeric parsers now allow arbitrary radix from 2..10 and 16

• The placeholder `_val` now can be used in top level semantic actions outside of any right hand side of a rule. Here is an example:

```c
int i = 0
BOOST_TEST(test_attr("456", int_[_val = _1], i) && i == 456);
```

In this case `_val` refers to the attribute value, which is passed in to the `parse()` or `phrase_parse()` functions. Similar constructs are now possible in Spirit.Karma as well:

```c
int i = 123;
BOOST_TEST(test("123", int_[_1 = _val], i));
```

This change unifies the handling of the `_val` placeholder allowing to use it everywhere, not only in semantic actions attached to the right hand sides of a rule.

• Added support for Spirit.Karma unsigned numeric generators with arbitrary radix values in the (inclusive) range from 2 .. 36.

Bug Fixes in Qi or Karma

• Spirit.Qi integer literals (like `int_(10)`) do not consume input on failure anymore.

• Fixed Boost ticket #5246: `mmap_file_iterator` Fails to initialize correctly.

• Fixed Boost ticket #5246: `mmap_file_iterator` Fails to initialize correctly.

• Fixed a const correctness problem in `karma::real_policies<>` preventing the use of const floating point types with the generator. Thanks to Jeroen Habraken (a.k.a. VeXocide) for reporting it and for submitting a patch and a test.
• Fixed the Spirit.Qi attr(attrib) parser, the Spirit.Qi symbols<> parser, and the Spirit.Karma symbols<> generator to properly handle container attributes. These were broken in Boost V1.46.1 (thanks to Aaron Graham and Joerg Becker for reporting the problems).

• Fixed the stream parser to properly adjust the iterator of the underlying input stream in the case when the stream parser component was successful. Thanks to Brian O’Kennedy who reported the problem on Stackoverflow.

• Fixed failing Karma numerics generators when used with adapted ADTs (thanks to Colin Rundel for reporting that problem).

Breaking Changes

• The Spirit.Qi directive repeat erroneously implemented commit/rollback semantics for its attribute, leaving it untouched if the directive failed. This behaviour has been removed as it is inconsistent with similar components. Existing code relying on this functionality will break. Please refer to the Spirit.Qi directive hold to see how to fix your code.

• Added a preprocessor define BOOST_SPIRIT_NO_PREDEFINED_TERMINALS to prevent instantiations of predefined terminal objects which slows down compile time a lot. When BOOST_SPIRIT_NO_PREDEFINED_TERMINALS is defined, the user instantiates the terminals that he needs. So, instead of writing using qi::uint_one writes instead: qi::uint_type uint_.

New Features in Lex

• Added the possibility to specify a token id while creating a token definition using lex::char_ and lex::string. Both primitives now accept a second parameter which will be interpreted as the requested token id for any token generated from this definition.

• Added a new token type lex::lexertl::position_token<>, which is essentially plug-in compatible with the existing lex::lexertl::token<> class. However it additionally stores the pair of iterators pointing to the underlying matched input sequence as an iterator_range.

Bug Fixes in Lex

• Fixed a problem with associating token definitions with all states (using *** as the state name) when actions were attached to them.

Making Stuff Work

• Added the Spirit.Karma customization point traits::extract_from_container, which will be invoked instead of the older customization point traits::extract_from if the attribute is a container (traits::is_container returns true for the attribute).

• The type hold_any now takes a template argument: basic_hold_any<Char>, where Char is the character type used for the streaming operators (operator>>() and operator<<(>). The hold_any is still available as a typedef basic_hold_any<char> hold_any;

• Semantic actions now support attribute compatibility. This is a breaking change but #define BOOST_SPIRIT_ACTIONS_ALLOW_ATTR_COMPAT must be defined in order for the new behavior to kick in. By default, the old behavior is still in place.

• Alternatives now support attribute compatibility.

• The attribute handling for container attributes of sequences and container components (list, Kleene, Plus, and repeat) has been completely rewritten. It now supports many more use cases and behaves much more predictable than the older version. Thanks to Thomas Taylor, Richard Crossley, Semen, Adalberto Castelo, and many others for reporting bugs and helping in making the new code behave as expected.
Spirit V2.4.2

What's changed in *Spirit.Qi* and *Spirit.Karma* from V2.4.1 (Boost V1.45.0) to V2.4.2 (Boost V1.46.0)

**New Features in Qi or Karma**

- Added keyword indexes for *Spirit.Qi* and *Spirit.Karma* to the docs.
- Introduced a new customization point `traits::assign_to_container_from_value` which is invoked for container attributes whenever an attribute value needs to be added to that container.
- Replaced `proto::lit` (which was used to implement `spirit::lit`) with a separate version allowing to distinguish 'lit(foo)' from 'foo'. This should not change any semantics nor should it break exiting code.
- Added the *Spirit.Qi* directive `as<T>[]` (and its string specializations `as_string[]` and `as_wstring[]`) enabling assignment of container attribute types as a whole.
- Added the *Spirit.Karma* directive `as<T>[]` (and its string specializations `as_string[]` and `as_wstring[]`) enabling handling of container attribute types during output generation as a whole.
- In *Spirit.Qi*, `lit()` can now be used for numeric literals as well.
- The `symbols<>` parser component now has an explicit name used for error handling and debugging, which can be set using the new member functions `sym.name(...)`. Thanks to teajay for contributing a patch.
- The `symbols<Attrib, T>` generator component now has an explicit name used for error handling and debugging, which can be set using the new member functions `sym.name(...)`.

**Bug Fixes in Qi or Karma**

- Fixed a problem in handling container attributes for *Spirit.Qi* sequences, which caused the properly parsed attributes of the first elements being overwritten by later elements of the sequence.
- Fixed the *Spirit.Karma* generator `string(s)`. It succeeded even if `s` matched only a prefix of its attribute.

What's changed in *Spirit.Lex* from V2.4.1 (Boost V1.45.0) to V2.4.2 (Boost V1.46.0)

**New Features in Lex**

- Added `qi::tokenid()` primitive parser allowing to match arbitrary lexer tokens based on a given token id. The difference to `qi::token()` is, that it exposes as its attribute the token id of the matched token (instead of the iterator_range of the matched input, as `qi::token()` does).
- Added an additional template parameter to the default `lexertl::token<>` definition: the type of the token id. This type defaults to `std::size_t`. Any type used as the id type needs to be (explicitly) convertible from `std::size_t`.
- It's now possible to attach lexer semantic actions to token definitions based on `lex::char()` and `lex::string()`.
- It's now possible to specify a lexer state the lexer should automatically be switched to after matching certain tokens. For this reason the token definition syntax has been extended:
template <typename Lexer>
struct lexer : lex::lexer<Lexer>
{
  lexer()
  {
    int_ = "[1-9][0-9]*";
    this->self("INITIAL", "TARGETSTATE") = int_
  }
  lex::token_def<int> int_;
};

This example lexer will match a \texttt{int\_} token and will switch the lexer to the state \texttt{"TARGETSTATE"} afterwards. If the second argument is not specified the lexer remains in the previous state (as before).

- The parser primitives \texttt{qi::token} and \texttt{qi::tokenid} can now be used without any argument. In this case they will match any token.
- \texttt{lex::lit()} has been removed.

**Bug Fixes in Lex**

- Fixed an issue in the Lexer giving problems while assigning tokens to all lexer states at once. This is now possible by simply using \texttt{"\*"} as the state name. For instance this will add the token \texttt{int\_} to all lexer states:

```cpp
template <typename Lexer>
struct lexer : lex::lexer<Lexer>
{
  lexer()
  {
    int_ = "[1-9][0-9]*";
    this->self("\*") = int_
  }
  lex::token_def<int> int_;
};
```

Note: the \texttt{self("\*") = ...} must be executed after all lexer states have been introduced to the lexer object.

- Fixed lexer lookahead. The lookahead operation is now evaluated using the lexer state the \texttt{token_def} instance used as its argument is associated with.
- Fixed a problem in the \texttt{multi_pass} iterator causing wrong tokens to be returned to the user. This could happen in conjunction with a lexer which performed lexer state changes and was using \texttt{pass_fail} in semantic actions to make a token match fail.

**Known Problems**

- \texttt{Spirit.Qi} integer literals (like \texttt{int\_}(10)) consume input on failure, which can lead to problems with the alternative operator.

**Spirit V2.4.1**

What's changed in \texttt{Spirit.Qi} and \texttt{Spirit.Karma} from V2.4 (Boost V1.44.0) to V2.4.1 (Boost V1.45.0)

**Bug Fixes**

- Fixed broken \texttt{Spirit.Qi} debugging facilities for optional attributes.
- The \texttt{Spirit.Qi} auto parsers and \texttt{Spirit.Karma} auto generators will now properly work with \texttt{signed char} and \texttt{unsigned char} as well.
- Fixed a problem in the \texttt{multi_pass fixed_queue} policy.
• Enabled proper modifier handling for the `skip` directive.

• Fixed a floating point formatting problem in Karma (Trac ticket #4742).

• Fixed a problem in `repeat[]`, which caused the `first` iterator not to be reset on certain parsing failures (see Stack Overflow).

**Spirit V2.4**

**What's changed in Spirit.Qi and Spirit.Karma from V2.3 (Boost V1.43.0) to V2.4 (Boost V1.44.0)**

**New Features**

• The customization point `traits::transform_attribute` now takes an additional template parameter `Domain` allowing to better specialize the customization point for either `qi::domain` or `karma::domain`.

  **Important**

  This is an interface breaking change requiring to modify existing code. If you have a specialization of this customization point in your code you need to add the specialization for the new template parameter, i.e. either `qi::domain` or `karma::domain`.

• Semantic actions in Spirit.Qi now implicitly invoke the function `pre` of the customization point `traits::transform_attribute` to convert the supplied attribute to the exposed attribute type, as needed. The functions `post` and `fail` of this customization point are not invoked by this component (as this would not make any sense).

• Semantic actions in Spirit.Karma now implicitly invoke the function `pre` of the customization point `traits::transform_attribute` to convert the supplied attribute to the consumed attribute type, as needed.

• Added the Spirit.Karma `skip` directive which is semantically equivalent to the Spirit.Karma `omit` directive except that it will not execute the embedded generator.

• Added debug support to Spirit.Karma rules.

• Added strict mode to Spirit.Karma, leaving the current behavior (unchanged) as relaxed mode. Added Spirit.Karma compile time directives `strict[]` and `relaxed[]` allowing to switch between the two.

• Added Spirit.Karma `duplicate` directive which duplicates the supplied attribute to all elements of an embedded generator sequence.

**Bug Fixes**

• Components in Spirit.Qi and Spirit.Karma now accept one element Fusion sequences as their attributes as long as the element in the Fusion sequence is compatible with the component's attribute type.

• The character range parser and generator components can now additionally be written as `char_("a", "z")` instead of `char_("a", 'z')` making it consistent with the syntax of the `char_("a")` component (which can be written as `char_("a")` as well). Please note that the mixed syntax forms, i.e. `char_("a", "z")` and `char_("a", 'z')`, are not supported.

• Fixed attribute handling in Spirit.Karma sequences when all elements of that sequence consume either the same attribute type or containers of that attribute type and the passed in attribute is a container of that attribute type as well. In this case using a repetitive container was supported only when it was the last element of the sequence. Now it is possible to have a `repeat(num)[a]` generator at any position (well, actually you can have any repetitive container at any position now, but this doesn't always make sense as it normally would eat up all supplied attribute values).

• Fixed debug output for variants where a variant element is an STL sequence.

• Fixed a problem in multi_pass, avoiding to loose a character at end of input when switching iterators.
What's changed in **Spirit.Lex** from V2.3 (Boost V1.43.0) to V2.4 (Boost V1.44.0)

**New Lexer Features**

- The lexer is now well integrated with the debug output generated by Qi’s simple_trace utility. Tokens are printed as: ‘<’ matched sequence ‘>’.

**Lexer Bug Fixes**

- Fixed a problem with using lex::_val as a rvalue in lexer semantic expressions.
- Token values are now available for introspection (as an iterator_range) inside lexer semantic expressions as well.

**Spirit V2.3**

What's changed in **Spirit.Qi** and **Spirit.Karma** from V2.2 (Boost V1.42.0) to V2.3 (Boost V1.43.0)

**New Features**

- The customization point `transform_attribute` now has to implement a third function: `void fail(Exposed&)`, which normally will do nothing. This function will be called whenever the right hand side of the rule (or the embedded parser of `attr_cast`) fail parsing. This change affects *Qi* only. See the description of the `traits::transform_attribute` for more details.
- Added support for attribute sequences created with `BOOST_FUSION_ADAPT_CLASS` and `BOOST_FUSION_ADAPT_CLASS_NAMED`. This support requires to include the new header file: `#include <boost/spirit/include/support_adapt_class_attributes.hpp>`.
- Added `karma::ostream_iterator` as a counterpart to `qi::istream_iterator` (see new header file: `#include <boost/spirit/home/support/iterators/ostream_iterator.hpp>`).
- Added `qi::hold` allowing to make sure the embedded parser does not touch the passed attribute in case it fails parsing.
- Added `qi::no_skip` directive, which is equivalent to `qi::lexeme`, except that it does not pre-skip.
- Added `karma::no_delimit` directive, which is equivalent to `karma::verbatim`, except that it does not perform a post-delimiting step.
- Added a new input_iterator policy for the `multi_pass` iterator framework (named `buffering_input_iterator`) allowing to wrap underlying input iterators which do not store the last character read from the input (such as `std::istream_iterator`). This is now used as the default input policy.

**Bug Fixes**

- Sequences (in *Qi* and *Karma*) may now have a component having no attribute even as their last element.
- Sequences (in *Qi* and *Karma*) can now take one element attribute sequences as their attribute.
- Constructs like `karma::buffer[karma::buffer[...]]` don’t result in performing double buffering anymore. The same is true if an alternative is wrapped into a `karma::buffer[]` directive (as for instance: `buffer[a] | b`).
- The *Spirit.Karma* output iterator (which is used internally, but also is exposed when using the stream based API) is now properly copyable (thanks to Jonas Persson for reporting this issue).
- The default `multi_pass` iterator is now usable with underlying input iterators which do not store the last character read from the input (such as `std::istream_iterator`). Thanks to Larry Evans and Peter Schueller for independently reporting this problem.
- The directive `karma::omit[]` now does not accept an arbitrary attribute type anymore.
- The *Spirit.Karma* predicates (the and-predicate and the not-predicate) and the directive `karma::omit[]` now disable output altogether instead of intercepting the output into a buffer which got discarded as before.
• Fixed `karma::rule` to properly handle optional attributes.

What's changed in *Spirit.Lex* from V2.2 (Boost V1.42.0) to V2.3 (Boost V1.43.0)

**New Lexer Features**

The library does not minimize the generated lexer tables for dynamic lexers by default anymore. The generated tables will now be minimized for static lexers only.

The function `lexer<>::init_dfa()` now takes a single boolean parameter (which defaults to `false`) allowing to force minimization of the generated lexer tables.

**Spirit V2.2**

What's changed in *Spirit.Qi* and *Spirit.Karma* from V2.1 (Boost V1.41.0) to V2.2 (Boost V1.42.0)

**New Features**

• Added `auto_` component in *Spirit.Qi* and *Spirit.Karma*, added API functions `qi::create_parser` and `karma::create_generator`.

• Added `auto_` based overloads for all API functions taking no attributes (see *Qi API* and *Karma API*).

• Added `karma::columns` directive.

• Added `karma::symbols<Attr, T>` generator.

• The *Spirit.Qi* customization point `traits::push_back_container` now returns a `bool` to report whether the item has been added to the container.

• Added an overload for `karma::maxwidth` directive allowing to specify an additional parameter (any compatible output iterator) receiving the 'overspilled' output (output not fitting into the maxwidth limit).

• It is now possible to use Phoenix expressions as *Spirit.Karma* attributes.

• Added `qi::matches` directive.

**Bug Fixes**

• Fixed `karma::alternatives` to work with embedded containers of `hold_any` (i.e. constructs like `*stream | "empty"` (which fixes the Karma example `basic_facilities.cpp`).

• Fixed numeric *Spirit.Karma* generators for character types.

• Fixed `qi::repeat[]` for unused attributes.

• Fixed rare compilation problem in `karma::repeat[]`.

• Fixed sequences in *Spirit.Qi* and *Spirit.Karma* to compile properly if the attribute is a (STL) container of (STL) containers.

• Fixed a problem in `lex::token_def::what`.

• Fixed *Spirit.Qi* symbols not to match substrings anymore. Added `qi::symbols::prefix_find` to allow matching of (prefix-) substrings.

• Inherited parameters for rule's usually have to be wrapped in function objects (i.e. `phoenix::val`), for integral values this was not necessary. Now all string types can be passed without being wrapped as well (i.e. `std::string`, `char const*`, etc.).
• Added concept checks to all relevant Spirit.Qi API functions enforcing the iterator to be at least of the type std::forward_iterator_tag.

• Fixed the qi::match and qi::phrase_match set of API functions not to internally utilize a std::stream_iterator anymore as this iterator is of the type std::input_iterator_tag only, which is not sufficient for Spirit.Qi.

Spirit V2.1

What's changed in Spirit.Qi and Spirit.Karma from V2.0 (Boost V1.37.0) to V2.1 (Boost V1.41.0)

• Spirit is now based on the newest version of Boost.Proto

• qi::phrase_parse, qi::phrase_format now post-skip by default.

• karma::generate_delimited and karma::format_delimited now don't do pre-delimiting by default.

• Changed parameter sequence of qi::phrase_parse, qi::phrase_match, karma::generate_delimited, and match_delimited. The attribute is now always the last parameter.

• Added new overloads of those functions allowing to explicitly specify the post-skipping and pre-delimiting behavior.

• Added multi attribute API functions

• Removed grammar_def<>

• Removed functions make_parser() and make_generator()

• Removed qi::none and karma::none

• Sequences and lists now accept a standard container as their attribute

• The string placeholder terminal now can take other strings as its parameter (i.e. std::string)

• All terminals taking literals now accept a (lazy) function object as well

• All placeholders for terminals and directives (such as int_, double_, verbatim, etc.) were previously defined in the namespace boost::spirit only. Now these are additionally imported into the namespaces spirit::qi, spirit::karma, and spirit::lex (if they are supported by the corresponding sub-library).

• The terminal placeholders char_ and string are not defined in the namespace boost::spirit anymore as they have been moved to the character set namesaces, allowing to do proper character set handling based on the used namespace (as spirit::ascii, etc.)

• The uint, ushort, ulong, and byte terminal placeholders have been renamed to uint_, ushort_, ulong_, and byte_

• qi::skip[] now re-enables outer skipper if used inside lexeme[]

• Added karma::maxwidth[] directive (see maxwidth)

• Added karma::omit[] allowing to consume the attribute of subject generator without emitting any output (see omit).

• Added karma::buffer[] allowing to avoid unwanted output to be generated in case of a generator failing in the middle of a sequence (see buffer).

• karma::delimit[] now re-enables outer delimiter if used inside verbatim[]

• Karma: added and-predicate (operator&()) and not-predicate (operator!()) Both now always consume an attribute.

• Karma: changed semantics of char_(), string(), int_() et.al., and double_() et.al.: all of these generators now always expose an attribute. If they do not have an associated attribute, they generate their immediate literal. If they have an associated
attribute, the generators first test if the attribute value is equal to the immediate literal. They fail and do not generate anything if those are not equal. Otherwise they generate their immediate literal. For more information see for instance \texttt{int_<}.

\begin{itemize}
  \item \texttt{karma::lit()} can now be used to generate integer and floating point numbers
  \item \texttt{qi::rule} and \texttt{karma::rule} now can be directly initialized using their copy constructor. I.e. this works now: \texttt{qi::rule<...> r = ...some parser...;}
  \item Added \texttt{qi::attr()} exposing its immediate parameter as its attribute.
  \item Added boolean parsers and generators (\texttt{bool_<, true_<, false_<}).
  \item Added \texttt{attr_cast<>} enabling in place attribute type conversion in Qi and Karma grammars.
  \item Almost all Karma generators now accept \texttt{optional<>} attributes and will fail generating if this is not initialized.
  \item Qi and Karma rules now automatically detect whether to apply auto-rule semantics or not (no need for using \texttt{operator\%=()} anymore, even if it’s still existing). Auto-rule semantics are applied if the right hand side has no semantic actions attached to any of the elements. This works for rule initialization and assignment.
  \item Qi and Karma rules now do intrinsic attribute transformation based on the attribute customization point \texttt{traits::transform_attribute}.
  \item All char\_< parsers now always expose an attribute. Earlier \texttt{char\_<(...) didn’t expose an attribute while char\_< did. If you need a literal parser not exposing any attribute use \texttt{lit()} instead.
  \item The \texttt{qi::int_spec, qi::real_spec, karma::int_spec, and karma real_spec types do not exist anymore. These have been replaced with qi::int_parser, qi::real_parser, karma::int_generator, and karma::real_generator.}
\end{itemize}

\textbf{What’s changed in \textit{Spirit.Lex} from V2.0 (Boost V1.37.0) to V2.1 (Boost V1.41.0)}

Here is a list of changes in \textit{Spirit.Lex} since version 2.0. \textit{Spirit.Lex} 2.1 is a complete rewrite of the \textit{Spirit.Lex} distributed with Boost V1.37. As with all code portions of the \textit{Spirit} library, \textit{Spirit.Lex} is usable as stand alone piece. \textit{Spirit.Lex} now uses the infrastructure provided by \textit{Spirit} version 2.1.

- The \texttt{lex:lexer_def class has been renamed to lex:lexer, while the original class lex:lexer does not exist anymore. This simplifies the creation of lexers.}
- The \texttt{lex:lexer class does not have the function \texttt{def(Self& self) anymore, token definitions can be added to the lexer at any time, usually in the constructor of the user defined lexer class:}

\begin{verbatim}
  template<typename Lexer>
  struct example_tokens : lex:lexer<Lexer>
  {
    example_tokens()
    {
      // your token definitions here
      this->self = ...
    }
  };
\end{verbatim}

- The new lexer class can now be used directly. The function \texttt{make_lexer()} has been removed.
- The \texttt{lex:tokenize_and_parse()} and \texttt{lex:tokenize_and_phrase_parse()} functions have been changed to match the parameter sequence as implemented by the \texttt{qi::parse()} and \texttt{qi::phrase_parse()} functions. Both take an arbitrary number of attribute arguments as the last parameters. This argument list is limited by the macro \texttt{SPIRIT_ARGUMENTS_LIMIT}.
- The \texttt{lex:lexertl_lexer, and lex:lexertl_token classes have been moved to the lex:lexertl namespace and the names have been changed to lex:lexertl_lexer, lex:lexertl_token. This also applies to the lex:lexert_act-
or_lexer, and the static_lexertl_* family of types.
• The class `lex::lexertl_token_set` has been removed. This functionality is now available from the lexer class.

• The *Spirit.Lex* library has been updated to use the newest version of Ben Hansons *Lexertl* lexer construction library (Boost review pending).

• The `lex::lexer<Lexer>` template constructor now takes an optional parameter specifying the `match_flags` to be used for table generation. Currently, there are the following flags available:

```
match_flags::match_default, // no flags
match_flags::match_not_dot_newline, // the regex '.' doesn't match newlines
match_flags::match_icase // all matching operations are case insensitive
```

If no parameter is passed to the constructor, `match_flags::match_default` is used, i.e. the . matches newlines and matching is case sensitive.

• The `char_()` and `string_()` placeholders can now be used for token definitions and are synonymous with `token_def`.

• Lexer semantic actions now have to conform to a changed interface (see Lexer Semantic Actions for details).

• Added placeholder symbols usable from the inside of lexer semantic actions while using Phoenix: `lex::_start, lex::_end, lex::_eoi, lex::_state, lex::_val, and lex::_pass` (see Lexer Semantic Actions for more details).

• Added (lazy) support functions usable from the inside of lexer semantic actions while using Phoenix: `lex::more(), lex::less()`, and `lex::lookahead()` (see Lexer Semantic Actions for more details).

• Removed `lex::omitted` in favor of `lex::omit` to unify the overall interface.

**Spirit Classic**

The Spirit V1.8.x code base has been integrated with Spirit V2. It is now called *Spirit Classic*. Since the directory structure has changed (the Spirit Classic headers are now moved to the `$BOOST_ROOT/boost/spirit/home/classic` directory), we created forwarding headers allowing existing applications to compile without any change. However, these forwarding headers are deprecated, which will result in corresponding warnings generated for each of the headers starting with Boost V1.38. The forwarding headers are expected to be removed in the future.

The recommended way of using Spirit Classic now is to include header files from the directory `$BOOST_ROOT/boost/spirit/include`. All Spirit Classic headers in this directory have `classic_` prefixed to their name. For example the include

```
#include <boost/spirit/core/core.hpp>
```

now should be written as:

```
#include <boost/spirit/include/classic_core.hpp>
```

To avoid namespace conflicts with the new Spirit V2 library we moved Spirit Classic into the namespace `boost::spirit::classic`. All references to the former namespace `boost::spirit` need to be adjusted as soon as the header names are corrected as described above. As an alternative you can define the preprocessor constant `BOOST_SPIRIT_USE_OLD_NAMESPACE`, which will force the Spirit Classic code to be in the namespace `boost::spirit` as before. This is not recommended, though, as it may result in naming clashes.

The change of the namespace will be automatically deactivated whenever the deprecated include files are being used. This ensures full backwards compatibility for existing applications.
Introduction

Boost Spirit is an object-oriented, recursive-descent parser and output generation library for C++. It allows you to write grammars and format descriptions using a format similar to Extended Backus Naur Form (EBNF)\(^2\) directly in C++. These inline grammar specifications can mix freely with other C++ code and, thanks to the generative power of C++ templates, are immediately executable. In retrospect, conventional compiler-compilers or parser-generators have to perform an additional translation step from the source EBNF code to C or C++ code.

The syntax and semantics of the libraries' API directly form domain-specific embedded languages (DSEL). In fact, Spirit exposes 3 different DSELS to the user:

- one for creating parser grammars,
- one for the specification of the required tokens to be used for parsing,
- and one for the description of the required output formats.

Since the target input grammars and output formats are written entirely in C++ we do not need any separate tools to compile, preprocess or integrate those into the build process. Spirit allows seamless integration of the parsing and output generation process with other C++ code. This often allows for simpler and more efficient code.

Both the created parsers and generators are fully attributed, which allows you to easily build and handle hierarchical data structures in memory. These data structures resemble the structure of the input data and can directly be used to generate arbitrarily-formatted output.

The figure below depicts the overall structure of the Boost Spirit library. The library consists of 4 major parts:

- **Spirit Classic**: This is the almost-unchanged code base taken from the former Boost Spirit V1.8 distribution. It has been moved into the namespace boost::spirit::classic. A special compatibility layer has been added to ensure complete compatibility with existing code using Spirit V1.8.

- **Spirit Qi**: This is the parser library allowing you to build recursive descent parsers. The exposed domain-specific language can be used to describe the grammars to implement, and the rules for storing the parsed information.

- **Spirit Lex**: This is the library usable to create tokenizers (lexers). The domain-specific language exposed by Spirit.Lex allows you to define regular expressions used to match tokens (create token definitions), associate these regular expressions with code to be executed whenever they are matched, and to add the token definitions to the lexical analyzer.

- **Spirit Karma**: This is the generator library allowing you to create code for recursive descent, data type-driven output formatting. The exposed domain-specific language is almost equivalent to the parser description language used in Spirit.Qi, except that it is used to describe the required output format to generate from a given data structure.

\(^2\) ISO-EBNF
Figure 1. The overall structure of the Boost Spirit library

The three components, Spirit.Qi, Spirit.Karma and Spirit.Lex, are designed to be used either stand alone, or together. The general methodology is to use the token sequence generated by Spirit.Lex as the input for a parser generated by Spirit.Qi. On the opposite side of the equation, the hierarchical data structures generated by Spirit.Qi are used for the output generators created using Spirit.Karma. However, there is nothing to stop you from using any of these components all by themselves.

The figure below shows the typical data flow of some input being converted to some internal representation. After some (optional) transformation these data are converted back into some different, external representation. The picture highlights Spirit's place in this data transformation flow.

Figure 2. The place of Spirit.Qi and Spirit.Karma in a data transformation flow of a typical application
A Quick Overview of Parsing with Spirit.Qi

Spirit.Qi is Spirit's sublibrary dealing with generating parsers based on a given target grammar (essentially a format description of the input data to read).

A simple EBNF grammar snippet:

```
group ::= '(' expression ')'
factor ::= integer | group
term ::= factor (('*' factor) | ('/' factor))*
expression ::= term (('+') term | ('-' term))*
```

is approximated using facilities of Spirit's Qi sublibrary as seen in this code snippet:

```
group = '(' >> expression >> ')';
factor = integer | group;
term = factor >> *((('*' >> factor) | ('/' >> factor));
expression = term >> *((('+') >> term) | ('-' >> term));
```

Through the magic of expression templates, this is perfectly valid and executable C++ code. The production rule `expression` is, in fact, an object that has a member function `parse` that does the work given a source code written in the grammar that we have just declared. Yes, it's a calculator. We shall simplify for now by skipping the type declarations and the definition of the rule `integer` invoked by `factor`. Now, the production rule `expression` in our grammar specification, traditionally called the start symbol, can recognize inputs such as:

```
12345
-12345
+12345
1 + 2
1 * 2
1/2 + 3/4
1 + 2 + 3 + 4
1 * 2 * 3 * 4
(1 + 2) * (3 + 4)
(-1 + 2) * (3 + -4)
1 + ((6 * 200) - 20) / 6
(1 + (2 + (3 + (4 + 5))))
```

Certainly we have modified the original EBNF syntax. This is done to conform to C++ syntax rules. Most notably we see the abundance of `shift >>` operators. Since there are no 'empty' operators in C++, it is simply not possible to write something like:

```
a b
```

as seen in math syntax, for example, to mean multiplication or, in our case, as seen in EBNF syntax to mean sequencing (b should follow a). Spirit.Qi uses the `shift >>` operator instead for this purpose. We take the `>>` operator, with arrows pointing to the right, to mean "is followed by". Thus we write:

```
a >> b
```

The alternative operator `|` and the parentheses `()` remain as is. The assignment operator `=` is used in place of EBNF's `::=`. Last but not least, the Kleene star `*`, which in this case is a postfix operator in EBNF becomes a prefix. Instead of:

```
a* //... in EBNF syntax,
```

we write:
since there are no postfix stars, *, in C/C++. Finally, we terminate each rule with the ubiquitous semi-colon, ;.

A Quick Overview of Output Generation with Spirit.Karma

Spirit not only allows you to describe the structure of the input, it also enables the specification of the output format for your data in a similar way, and based on a single syntax and compatible semantics.

Let's assume we need to generate a textual representation from a simple data structure such as a `std::vector<int>`. Conventional code probably would look like:

```cpp
std::vector<int> v (initialize_and_fill());
std::vector<int>::iterator end = v.end();
for (std::vector<int>::iterator it = v.begin(); it != end; ++it)
    std::cout << *it << std::endl;
```

which is not very flexible and quite difficult to maintain when it comes to changing the required output format. Spirit's sublibrary Karma allows you to specify output formats for arbitrary data structures in a very flexible way. The following snippet is the Karma format description used to create the same output as the traditional code above:

```
*(int_ << eol)
```

Here are some more examples of format descriptions for different output representations of the same `std::vector<int>`:

<table>
<thead>
<tr>
<th>Format</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'[' &lt;&lt; *(int_ &lt;&lt; ',') &lt;&lt; ']'</td>
<td>[1,8,10,]</td>
<td>Comma separated list of integers</td>
</tr>
<tr>
<td>*((',' &lt;&lt; int_ &lt;&lt; ',') &lt;&lt; ',')</td>
<td>(1),(8),(10),</td>
<td>Comma separated list of integers in parenthesis</td>
</tr>
<tr>
<td>*hex</td>
<td>18a</td>
<td>A list of hexadecimal numbers</td>
</tr>
<tr>
<td>*(double_ &lt;&lt; ',')</td>
<td>1.0,8.0,10.0,</td>
<td>A list of floating point numbers</td>
</tr>
</tbody>
</table>

We will see later in this documentation how it is possible to avoid printing the trailing ',', '.

Overall, the syntax is similar to Spirit.Qi with the exception that we use the << operator for output concatenation. This should be easy to understand as it follows the conventions used in the Standard's I/O streams.

Another important feature of Spirit.Karma allows you to fully decouple the data type from the output format. You can use the same output format with different data types as long as these conform conceptually. The next table gives some related examples.
Table 3. Different data types usable with the output format `*(int_ << eol)`

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i[4]</td>
<td>C style arrays</td>
</tr>
<tr>
<td>std::vector&lt;int&gt;</td>
<td>Standard vector</td>
</tr>
<tr>
<td>std::list&lt;int&gt;</td>
<td>Standard list</td>
</tr>
<tr>
<td>boost::array&lt;long, 20&gt;</td>
<td>Boost array</td>
</tr>
</tbody>
</table>
**Structure**

**Include**

Spirit is a header file only library. There are no libraries to link to. This section documents the structure of the Spirit headers.

Spirit contains five sub-libraries plus a 'support' module where common support classes are placed:

- Classic
- Qi
- Karma
- Lex
- Phoenix
- Support

The top Spirit directory is:

```
BOOST_ROOT/boost/spirit
```

Currently, the directory contains:

```
[actor] [attribute] [core] [debug]  
[dynamic] [error_handling] [home] [support]  
[iterator] [meta] [phoenix] [repository]  
[symbols] [tree] [utility]  
```

These include some old v1.8 directories that are now deprecated. These are: actor, attribute, core, debug, dynamic, error_handling, iterator, meta, phoenix, symbols, tree and utility. There is no guarantee that these directories will still be present in future versions of Spirit. We only keep them for backward compatibility. Please be warned.

Each directory (except include, home, and repository) has a corresponding header file that contains forwarding includes of each relevant include file that the directory contains. For example, there exists a `<boost/spirit/actor.hpp>` header file which includes all the relevant files from the boost/spirit/actor directory.

To distinguish between Spirit versions, you can inspect the version file:

```
<boost/spirit/version.hpp>
```

using the preprocessor define

```
SPIRIT_VERSION
```

It is a hex number where the first two digits determine the major version while the last two digits determine the minor version. For example:

```
#define SPIRIT_VERSION 0x2010 // version 2.1
```

The include directory at:
is a special flat directory that contains all the Spirit headers. To accommodate the flat structure, the headers are prefixed with the sub-library name:

- classic_
- karma_
- lex_
- phoenix1_
- phoenix_
- qi_
- support_

For example, if you used to include `<boost/spirit/actor.hpp>`, which is now a deprecated header, you should instead include `<boost/spirit/include/classic_actor.hpp>`

If you want to simply include the main sub-library name, then you can include:

- `<boost/spirit/include/classic.hpp>`
- `<boost/spirit/include/karma.hpp>`
- `<boost/spirit/include/lex.hpp>`
- `<boost/spirit/include/phoenix1.hpp>`
- `<boost/spirit/include/phoenix.hpp>`
- `<boost/spirit/include/qi.hpp>`
- `<boost/spirit/include/support.hpp>`

The home directory:

is the real home of Spirit. It is the place where the various sub-libraries actually exist. The home directory contains:

As usual, these directories have their corresponding include files:

- `<boost/spirit/home/classic.hpp>`
- `<boost/spirit/home/karma.hpp>`
- `<boost/spirit/home/lex.hpp>`
- `<boost/spirit/home/phoenix1.hpp>`
- `<boost/spirit/home/phoenix.hpp>`
- `<boost/spirit/home/qi.hpp>`
- `<boost/spirit/home/support.hpp>`
The various sub-libraries include files can be found in each sub-directory containing the particular sub-library. The include structure of a sub-library is covered in its documentation. For consistency, each library follows the same scheme as above.

To keep it simple, you should use the flat include directory at boost/spirit/include.

For some additional information about the rationale you might want to have a look at the FAQ entry Header Hell.

The subdirectory boost/spirit/repository does not belong to the main Spirit distribution. For more information please refer to: Spirit Repository.
Abstracts

Syntax Diagram

In the next section, we will deal with Parsing Expression Grammars (PEG)\(^3\), a variant of Extended Backus-Naur Form (EBNF)\(^4\) with a different interpretation. It is easier to understand PEG using Syntax Diagrams. Syntax diagrams represent a grammar graphically. It was used extensively by Niklaus Wirth\(^5\) in the "Pascal User Manual". Syntax Diagrams are easily understandable by programmers due to their similarity to flow charts. The isomorphism of the diagrams and functions make them ideal for representing Recursive Descent parsers which are essentially mutually recursive functions.

Historically, Parsing Expression Grammars have been used for describing grammars for parsers only (hence the name). In Spirit we use a very similar notation for output generation as well. Almost all the concepts described here are equally applicable both to Spirit.Qi parsers and to Spirit.Karma generators.

Elements

All diagrams have one entry and one exit point. Arrows connect all possible paths through the grammar from the entry point to the exit point.

![Start/Stop

Terminals are represented by round boxes. Terminals are atomic and usually represent plain characters, strings or tokens.

![Terminal

Non-terminals are represented by boxes. Diagrams are modularized using named non-terminals. A complex diagram can be broken down into a set of non-terminals. Non-terminals also allow recursion (i.e. a non-terminal can call itself).

![Non Terminal

Constructs

The most basic composition is the Sequence. B follows A:

![Sequence

The ordered choice henceforth we will call *alternatives*. In PEG, ordered choice and alternatives are not quite the same. PEG allows ambiguity of choice where one or more branches can succeed. In PEG, in case of ambiguity, the first one always wins.

---


\(^5\) Niklaus Wirth: The Programming Language Pascal. (July 1973)
The optional (zero-or-one):

Now, the loops. We have the zero-or-more and one-or-more:

Take note that, as in PEG, these loops behave greedily. If there is another 'A' just before the end-point, it will always fail because the preceding loop has already exhausted all 'A's and there is nothing more left. This is a crucial difference between PEG and general Context Free Grammars (CFGs). This behavior is quite obvious with syntax diagrams as they resemble flow-charts.

**Predicates**

Now, the following are Syntax Diagram versions of PEG predicates. These are not traditionally found in Syntax Diagrams. These are special extensions we invented to closely follow PEGs.

First, we introduce a new element, the Predicate:

This is similar to the conditionals in flow charts where the 'No' branch is absent and always signals a failed parse.

We have two versions of the predicate, the *And-Predicate* and the *Not-Predicate*:
The *And-Predicate* tries the predicate, P, and succeeds if P succeeds, or otherwise fail. The opposite is true with the *Not-Predicate*. It tries the predicate, P, and fails if P succeeds, or otherwise succeeds. Both versions do a look-ahead but do not consume any input regardless if P succeeds or not.

**Parsing Expression Grammar**

Parsing Expression Grammars (PEG) are a derivative of Extended Backus-Naur Form (EBNF) with a different interpretation, designed to represent a recursive descent parser. A PEG can be directly represented as a recursive-descent parser.

Like EBNF, PEG is a formal grammar for describing a formal language in terms of a set of rules used to recognize strings of this language. Unlike EBNF, PEGs have an exact interpretation. There is only one valid parse tree (see Abstract Syntax Tree) for each PEG grammar.

**Sequences**

Sequences are represented by juxtaposition like in EBNF:

```
a b
```

The PEG expression above states that, in order for this to succeed, `b` must follow `a`. Here's the syntax diagram:

```
A B
```

Here's a trivial example:

```
'x' digit
```

which means the character `x` must be followed by a digit.

**Note**

In *Spirit.Qi*, we use the `>>` for sequences since C++ does not allow juxtaposition, while in *Spirit.Karma* we use the `<<` instead.

**Alternatives**

Alternatives are represented in PEG using the slash:


7 Richard E. Pattis: EBNF: A Notation to Describe Syntax, [http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf](http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf)
Alternatives allow for choices. The expression above reads: try to match $a$. If $a$ succeeds, success, if not try to match $b$. This is a bit of a deviation from the usual EBNF interpretation where you simply match $a$ or $b$. Here's the syntax diagram:

PEGs allow for ambiguity in the alternatives. In the expression above, both $a$ or $b$ can both match an input string. However, only the first matching alternative is valid. As noted, there can only be one valid parse tree.

Loops

Again, like EBNF, PEG uses the regular-expression Kleene star and the plus loops:

$$a^*$$
$$a^+$$

Here are the syntax diagrams:

The first, called the Kleene star, matches zero or more of its subject $a$. The second, plus, matches one ore more of its subject $a$.

Unlike EBNF, PEGs have greedy loops. It will match as much as it can until its subject fails to match without regard to what follows. The following is a classic example of a fairly common EBNF/regex expression failing to match in PEG:
In PEG, alnum will eat as much alpha-numeric characters as it can leaving nothing more left behind. Thus, the trailing digit will get nothing. Loops are simply implemented in recursive descent code as for/while loops making them extremely efficient. That is a definite advantage. On the other hand, those who are familiar with EBNF and regex behavior might find the behavior a major gotcha. PEG provides a couple of other mechanisms to circumvent this. We will see more of these other mechanisms shortly.

**Difference**

In some cases, you may want to restrict a certain expression. You can think of a PEG expression as a match for a potentially infinite set of strings. The difference operator allows you to restrict this set:

```
  a - b
```

The expression reads: match a but not b.

### Note

There is no difference operator in *Spirit.Karma*, as the concept does not make sense in the context of output generation.

**Attributes**

### Attributes of Primitive Components

Parsers and generators in *Spirit* are fully attributed. *Spirit.Qi* parsers always expose an attribute specific to their type. This is called synthesized attribute as it is returned from a successful match representing the matched input sequence. For instance, numeric parsers, such as int_ or double_, return the int or double value converted from the matched input sequence. Other primitive parser components have other intuitive attribute types, such as for instance int_ which has int, or ascii::char_ which has char. For primitive parsers apply the normal C++ convertibility rules: you can use any C++ type to receive the parsed value as long as the attribute type of the parser is convertible to the type provided. The following example shows how a synthesized parser attribute (the int value) is extracted by calling the API function qi::parse:

```
int value = 0;
std::string str("123");
std::string::iterator strbegin = str.begin();
qi::parse(strbegin, str.end(), int_, value);  // value == 123
```

The attribute type of a generator defines what data types this generator is able to consume in order to produce its output. *Spirit.Karma* generators always expect an attribute specific to their type. This is called consumed attribute and is expected to be passed to the generator. The consumed attribute is most of the time the value the generator is designed to emit output for. For primitive generators the normal C++ convertibility rules apply. Any data type convertible to the attribute type of a primitive generator can be used to provide the data to generate. We present a similar example as above, this time the consumed attribute of the int_ generator (the int value) is passed to the API function karma::generate:

```
int value = 123;
std::string str;
std::back_insert_iterator<std::string> out(str);
karma::generate(out, int_, value);  // str == "123"
```

Other primitive generator components have other intuitive attribute types, very similar to the corresponding parser components. For instance, the ascii::char_ generator has char as consumed attribute. For a full list of available parser and generator primitives and their attribute types please see the sections *Qi Parsers* and *Karma Generators*.
Attributes of Compound Components

*Spirit.Qi* and *Spirit.Karma* implement well defined attribute type propagation rules for all compound parsers and generators, such as sequences, alternatives, Kleene star, etc. The main attribute propagation rule for a sequences is for instance:

<table>
<thead>
<tr>
<th>Library</th>
<th>Sequence attribute propagation rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qi</td>
<td>( a: A, b: B \rightarrow (a &gt;&gt; b): \text{tuple}&lt;A, B&gt; )</td>
</tr>
<tr>
<td>Karma</td>
<td>( a: A, b: B \rightarrow (a &lt;&lt; b): \text{tuple}&lt;A, B&gt; )</td>
</tr>
</tbody>
</table>

which reads as:

Given \( a \) and \( b \) are parsers (generators), and \( A \) is the attribute type of \( a \), and \( B \) is the attribute type of \( b \), then the attribute type of \( a >> b \) (\( a << b \)) will be \text{tuple}<A, B>.

**Note**

The notation \text{tuple}<A, B> is used as a placeholder expression for any fusion sequence holding the types \( A \) and \( B \), such as \text{boost::fusion::tuple}<A, B> or \text{std::pair}<A, B> (for more information see *Boost.Fusion*).

As you can see, in order for a type to be compatible with the attribute type of a compound expression it has to

- either be convertible to the attribute type,
- or it has to expose certain functionalities, i.e. it needs to conform to a concept compatible with the component.

Each compound component implements its own set of attribute propagation rules. For a full list of how the different compound generators consume attributes see the sections Parser Compound Attribute Rules and Generator Compound Attribute Rules.

The Attribute of Sequence Parsers and Generators

Sequences require an attribute type to expose the concept of a fusion sequence, where all elements of that fusion sequence have to be compatible with the corresponding element of the component sequence. For example, the expression:

<table>
<thead>
<tr>
<th>Library</th>
<th>Sequence expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qi</td>
<td>\text{double} &gt;&gt; \text{double}</td>
</tr>
<tr>
<td>Karma</td>
<td>\text{double} &lt;&lt; \text{double}</td>
</tr>
</tbody>
</table>

is compatible with any fusion sequence holding two types, where both types have to be compatible with \text{double}. The first element of the fusion sequence has to be compatible with the attribute of the first \text{double}, and the second element of the fusion sequence has to be compatible with the attribute of the second \text{double}. If we assume to have an instance of \text{std::pair}<\text{double}, \text{double}>, we can directly use the expressions above to do both, parse input to fill the attribute:

```cpp
// the following parses "1.0 2.0" into a pair of double
std::string input("1.0 2.0");
std::string::iterator strbegin = input.begin();
std::pair<double, double> p;
qi::phrase_parse(strbegin, input.end(),
qi::double_ >> qi::double_, // parser grammar
qi::space,                  // delimiter grammar
p);                          // attribute to fill while parsing
```

and generate output for it:
The following generates: "1.0 2.0" from the pair filled above
std::string str;
std::back_insert_iterator<std::string> out(str);
karma::generate_delimited(out,
    karma::double_ << karma::double_, // generator grammar (format description)
    karma::space, // delimiter grammar
    p); // data to use as the attribute
(where the karma::space generator is used as the delimiter, allowing to automatically skip/insert delimiting spaces in between all primitives).

Tip

For sequences only: Spirit.Qi and Spirit.Karma expose a set of API functions usable mainly with sequences. Very much like the functions of the scanf and printf families these functions allow to pass the attributes for each of the elements of the sequence separately. Using the corresponding overload of Qi's parse or Karma's generate()
the expression above could be rewritten as:

```c++
double d1 = 0.0, d2 = 0.0;
qi::phrase_parse(begin, end, qi::double_ >> qi::double_, qi::space, d1, d2);
karma::generate_delimited(out, karma::double_ << karma::double_, karma::space, d1, d2);
```

where the first attribute is used for the first double_, and the second attribute is used for the second double_.

The Attribute of Alternative Parsers and Generators

Alternative parsers and generators are all about - well - alternatives. In order to store possibly different result (attribute) types from the different alternatives we use the data type Boost.Variant. The main attribute propagation rule of these components is:

```
a: A, b: B --> (a | b): variant<A, B>
```

Alternatives have a second very important attribute propagation rule:

```
a: A, b: A --> (a | b): A
```

often allowing to simplify things significantly. If all sub expressions of an alternative expose the same attribute type, the overall alternative will expose exactly the same attribute type as well.

More About Attributes of Compound Components

While parsing input or generating output it is often desirable to combine some constant elements with variable parts. For instance, let us look at the example of parsing or formatting a complex number, which is written as (real, imag), where real and imag are the variables representing the real and imaginary parts of our complex number. This can be achieved by writing:

```
<table>
<thead>
<tr>
<th>Library</th>
<th>Sequence expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qi</td>
<td>'({ ' &gt;&gt; double_ &gt;&gt; &quot;, &quot; &gt;&gt; double_ &gt;&gt; '})'</td>
</tr>
<tr>
<td>Karma</td>
<td>'({ ' &lt;&lt; double_ &lt;&lt; &quot;, &quot; &lt;&lt; double_ &lt;&lt; '})'</td>
</tr>
</tbody>
</table>
```

Fortunately, literals (such as '{', '}', and ',') do not expose any attribute (well actually, they do expose the special type unused_type, but in this context unused_type is interpreted as if the component does not expose any attribute at all). It is very important to understand that the literals don't consume any of the elements of a fusion sequence passed to this component sequence. As said, they just don't expose any attribute and don't produce (consume) any data. The following example shows this:
// the following parses "(1.0, 2.0)" into a pair of double
std::string input("(1.0, 2.0)");
std::pair<double, double> p;
qi::parse(strbegin, input.end(),
    '(' >> qi::double_ >> ", " >> qi::double_ >> ')', // parser grammar
    p); // attribute to fill while parsing

and here is the equivalent Spirit.Karma code snippet:

// the following generates: (1.0, 2.0)
std::string str;
std::back_insert_iterator<std::string> out(str);
generate(out,
    '(' << karma::double_ << ", " << karma::double_ << ')', // generator grammar (format description)
    p); // data to use as the attribute

where the first element of the pair passed in as the data to generate is still associated with the first double_, and the second element is associated with the second double_ generator.

This behavior should be familiar as it conforms to the way other input and output formatting libraries such as scanf, printf or boost::format are handling their variable parts. In this context you can think about Spirit.Qi's and Spirit.Karma's primitive components (such as the double_ above) as of being type safe placeholders for the attribute values.

**Tip**

Similarly to the tip provided above, this example could be rewritten using Spirit's multi-attribute API function:

    double d1 = 0.0, d2 = 0.0;
    qi::parse(begin, end, '(' >> qi::double_ >> ", " >> qi::double_ >> ')', d1, d2);
    karma::generate(out, '(' << karma::double_ << ", " << karma::double_ << ')', d1, d2);

which provides a clear and comfortable syntax, more similar to the placeholder based syntax as exposed by printf or boost::format.

Let's take a look at this from a more formal perspective. The sequence attribute propagation rules define a special behavior if generators exposing unused_type as their attribute are involved (see Generator Compound Attribute Rules):

<table>
<thead>
<tr>
<th>Library</th>
<th>Sequence attribute propagation rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qi</td>
<td>a: A, b: Unused --&gt; (a &gt;&gt; b): A</td>
</tr>
<tr>
<td>Karma</td>
<td>a: A, b: Unused --&gt; (a &lt;&lt; b): A</td>
</tr>
</tbody>
</table>

which reads as:

Given a and b are parsers (generators), and A is the attribute type of a, and unused_type is the attribute type of b, then the attribute type of a >> b (a << b) will be A as well. This rule applies regardless of the position the element exposing the unused_type is at.

This rule is the key to the understanding of the attribute handling in sequences as soon as literals are involved. It is as if elements with unused_type attributes ‘disappeared’ during attribute propagation. Notably, this is not only true for sequences but for any compound components. For instance, for alternative components the corresponding rule is:
again, allowing to simplify the overall attribute type of an expression.

### Attributes of Rules and Grammars

Nonterminals are well known from parsers where they are used as the main means of constructing more complex parsers out of simpler ones. The nonterminals in the parser world are very similar to functions in an imperative programming language. They can be used to encapsulate parser expressions for a particular input sequence. After being defined, the nonterminals can be used as 'normal' parsers in more complex expressions whenever the encapsulated input needs to be recognized. Parser nonterminals in *Spirit.Qi* may accept *parameters* (inherited attributes) and usually return a value (the synthesized attribute).

Both, the types of the inherited and the synthesized attributes have to be explicitly specified while defining the particular grammar or the rule (the Spirit Repository additionally has subrules which conform to a similar interface). As an example, the following code declares a *Spirit.Qi* rule exposing an int as its synthesized attribute, while expecting a single double as its inherited attribute (see the section about the *Spirit.Qi* Rule for more information):

```cpp
qi::rule<Iterator, int(double)> r;
```

In the world of generators, nonterminals are just as useful as in the parser world. Generator nonterminals encapsulate a format description for a particular data type, and, whenever we need to emit output for this data type, the corresponding nonterminal is invoked in a similar way as the predefined Spirit.Karma generator primitives. The *Spirit.Karma* nonterminals are very similar to the *Spirit.Qi* nonterminals. Generator nonterminals may accept *parameters* as well, and we call those inherited attributes too. The main difference is that they do not expose a synthesized attribute (as parsers do), but they require a special *consumed attribute*. Usually the consumed attribute is the value the generator creates its output from. Even if the consumed attribute is not 'returned' from the generator we chose to use the same function style declaration syntax as used in *Spirit.Qi*. The example below declares a *Spirit.Karma* rule consuming a double while not expecting any additional inherited attributes.

```cpp
karma::rule<OutputIterator, double()> r;
```

The inherited attributes of nonterminal parsers and generators are normally passed to the component during its invocation. These are the *parameters* the parser or generator may accept and they can be used to parameterize the component depending on the context they are invoked from.
Why would you want to use Spirit.Qi?

Spirit.Qi is designed to be a practical parsing tool. The ability to generate a fully-working parser from a formal EBNF specification inlined in C++ significantly reduces development time. Programmers typically approach parsing using ad hoc hacks with primitive tools such as scanf. Even regular-expression libraries (such as boost regex) or scanners (such as Boost tokenizer) do not scale well when we need to write more elaborate parsers. Attempting to write even a moderately-complex parser using these tools leads to code that is hard to understand and maintain.

One prime objective is to make the tool easy to use. When one thinks of a parser generator, the usual reaction is "it must be big and complex with a steep learning curve." Not so. Spirit is designed to be fully scalable. The library is structured in layers. This permits learning on an as-needed basis, after only learning the minimal core and basic concepts.

For development simplicity and ease in deployment, the entire library consists of only header files, with no libraries to link against or build. Just put the Spirit distribution in your include path, compile and run. Code size? - very tight - essentially comparable to hand written recursive descent code.

Our tutorials will walk you through the simplest Spirit examples, incrementally building on top of the earlier examples as we expose more and more features and techniques. We will try to be as gentle as possible with the learning curve. We will present the tutorials in a cookbook style approach. This style of presentation is based on our BoostCon '07 and BoostCon '08 slides.

Have fun!

Warming up

We'll start by showing examples of parser expressions to give you a feel on how to build parsers from the simplest parser, building up as we go. When comparing EBNF to Spirit, the expressions may seem awkward at first. Spirit heavily uses operator overloading to accomplish its magic.

Trivial Example #1 Parsing a number

Create a parser that will parse a floating-point number.

```cpp
double_
```

(You've got to admit, that's trivial!) The above code actually generates a Spirit floating point parser (a built-in parser). Spirit has many pre-defined parsers and consistent naming conventions help you keep from going insane!

Trivial Example #2 Parsing two numbers

Create a parser that will accept a line consisting of two floating-point numbers.

```cpp
double_ >> double_
```

Here you see the familiar floating-point numeric parser `double_` used twice, once for each number. What's that `>>` operator doing in there? Well, they had to be separated by something, and this was chosen as the "followed by" sequence operator. The above program creates a parser from two simpler parsers, gluing them together with the sequence operator. The result is a parser that is a composition of smaller parsers. Whitespace between numbers can implicitly be consumed depending on how the parser is invoked (see below).
Note

When we combine parsers, we end up with a "bigger" parser, but it's still a parser. Parsers can get bigger and bigger, nesting more and more, but whenever you glue two parsers together, you end up with one bigger parser. This is an important concept.

Trivial Example #3 Parsing zero or more numbers

Create a parser that will accept zero or more floating-point numbers.

`*double_`

This is like a regular-expression Kleene Star, though the syntax might look a bit odd for a C++ programmer not used to seeing the `*` operator overloaded like this. Actually, if you know regular expressions it may look odd too since the star is before the expression it modifies. C'est la vie. Blame it on the fact that we must work with the syntax rules of C++.

Any expression that evaluates to a parser may be used with the Kleene Star. Keep in mind that C++ operator precedence rules may require you to put expressions in parentheses for complex expressions. The Kleene Star is also known as a Kleene Closure, but we call it the Star in most places.

Trivial Example #4 Parsing a comma-delimited list of numbers

This example will create a parser that accepts a comma-delimited list of numbers.

`double_ >> *(char_(',') >> double_)`

Notice `char_(',')`. It is a literal character parser that can recognize the comma ',', '. In this case, the Kleene Star is modifying a more complex parser, namely, the one generated by the expression:

`(char_(',') >> double_)`

Note that this is a case where the parentheses are necessary. The Kleene star encloses the complete expression above.

Let's Parse!

We're done with defining the parser. So the next step is now invoking this parser to do its work. There are a couple of ways to do this. For now, we will use the `phrase_parse` function. One overload of this function accepts four arguments:

1. An iterator pointing to the start of the input
2. An iterator pointing to one past the end of the input
3. The parser object
4. Another parser called the skip parser

In our example, we wish to skip spaces and tabs. Another parser named `space` is included in Spirit's repertoire of predefined parsers. It is a very simple parser that simply recognizes whitespace. We will use `space` as our skip parser. The skip parser is the one responsible for skipping characters in between parser elements such as the `double_` and `char_`.

Ok, so now let's parse!
template <typename Iterator>
bool parse_numbers(Iterator first, Iterator last)
{
    using qi::double_;  // Using qi::double_ to parse doubles
    using qi::phrase_parse;  // Using qi::phrase_parse for parsing phrases
    using ascii::space;  // Using ascii::space for skipping spaces

    bool r = phrase_parse(
        first,  // Start iterator
        last,  // End iterator
        double_ >> *(',' >> double_),  // The parser
        space  // The skip-parser
    );
    if (first != last) // fail if we did not get a full match
        return false;
    return r;  // The parser function returns true or false depending on the result of the parse. The first iterator is passed by reference. On a successful parse, this iterator is repositioned to the rightmost position consumed by the parser. If this becomes equal to last, then we have a full match. If not, then we have a partial match. A partial match happens when the parser is only able to parse a portion of the input.

    Note that we inlined the parser directly in the call to parse. Upon calling parse, the expression evaluates into a temporary, unnamed parser which is passed into the parse() function, used, and then destroyed.

    Here, we opted to make the parser generic by making it a template, parameterized by the iterator type. By doing so, it can take in data coming from any STL conforming sequence as long as the iterators conform to a forward iterator.

    You can find the full cpp file here: ../../example/qi/num_list1.cpp

    The parse function returns true or false depending on the result of the parse. The first iterator is passed by reference. On a successful parse, this iterator is repositioned to the rightmost position consumed by the parser. If this becomes equal to last, then we have a full match. If not, then we have a partial match. A partial match happens when the parser is only able to parse a portion of the input.

    Note that we inlined the parser directly in the call to parse. Upon calling parse, the expression evaluates into a temporary, unnamed parser which is passed into the parse() function, used, and then destroyed.

    Here, we opted to make the parser generic by making it a template, parameterized by the iterator type. By doing so, it can take in data coming from any STL conforming sequence as long as the iterators conform to a forward iterator.

    You can find the full cpp file here: ../../example/qi/num_list1.cpp

    The careful reader may notice that the parser expression has ',' instead of char_(',') as the previous examples did. This is ok due to C++ syntax rules of conversion. There are >> operators that are overloaded to accept a char or wchar_t argument on its left or right (but not both). An operator may be overloaded if at least one of its parameters is a user-defined type. In this case, the double_ is the 2nd argument to operator>>, and so the proper overload of >> is used, converting ',' into a character literal parser.

    The problem with omitting the char_ should be obvious: 'a' >> 'b' is not a spirit parser, it is a numeric expression, right-shifting the ASCII (or another encoding) value of 'a' by the ASCII value of 'b'. However, both char_('a') >> 'b' and 'a' >> char_('b') are Spirit sequence parsers for the letter 'a' followed by 'b'. You'll get used to it, sooner or later.

    Finally, take note that we test for a full match (i.e. the parser fully parsed the input) by checking if the first iterator, after parsing, is equal to the end iterator. You may strike out this part if partial matches are to be allowed.
Parser Semantic Actions

The example in the previous section was very simplistic. It only recognized data, but did nothing with it. It answered the question: "Did the input match?". Now, we want to extract information from what was parsed. For example, we would want to store the parsed number after a successful match. To do this, you will need *semantic actions*.

Semantic actions may be attached to any point in the grammar specification. These actions are C++ functions or function objects that are called whenever a part of the parser successfully recognizes a portion of the input. Say you have a parser $P$, and a C++ function $F$. You can make the parser call $F$ whenever it matches an input by attaching $F$:

$$P[F]$$

The expression above links $F$ to the parser, $P$.

The function/function object signature depends on the type of the parser to which it is attached. The parser `double_` passes the parsed number. Thus, if we were to attach a function $F$ to `double_`, we need $F$ to be declared as:

```cpp
void F(double n);
```

There are actually 2 more arguments being passed (the parser context and a reference to a boolean 'hit' parameter). We don't need these, for now, but we'll see more on these other arguments later. Spirit.Qi allows us to bind a single argument function, like above. The other arguments are simply ignored.

**Examples of Semantic Actions**

Presented are various ways to attach semantic actions:

- Using plain function pointer
- Using simple function object
- Using `Boost.Bind` with a plain function
- Using `Boost.Bind` with a member function
- Using `Boost.Lambda`

Given:
namespace client
{
    namespace qi = boost::spirit::qi;

    // A plain function
    void print(int const& i)
    {
        std::cout << i << std::endl;
    }

    // A member function
    struct writer
    {
        void print(int const& i) const
        {
            std::cout << i << std::endl;
        }
    };

    // A function object
    struct print_action
    {
        void operator()(int const& i, qi::unused_type, qi::unused_type) const
        {
            std::cout << i << std::endl;
        }
    };
}

Take note that with function objects, we need to have an `operator()` with 3 arguments. Since we don't care about the other two, we can use `unused_type` for these. We'll see more of `unused_type` elsewhere. `unused_type` is a Spirit supplied support class.

All examples parse inputs of the form:

```
"{integer}"
```

An integer inside the curly braces.

The first example shows how to attach a plain function:

```cpp
parse(first, last, '{' >> int_([&print]) >> '}');
```

What's new? Well `int_` is the sibling of `double_`. I'm sure you can guess what this parser does.

The next example shows how to attach a simple function object:

```cpp
parse(first, last, '{' >> int_([print_action()]) >> '}');
```

We can use `Boost.Bind` to 'bind' member functions:

```cpp
writer w;
parsed(first, last, '{' >> int_([boost::bind(&writer::print, &w, _1)]) >> '}');
```

Likewise, we can also use `Boost.Bind` to 'bind' plain functions:
Yep, we can also use Boost.Lambda:

```cpp
parse(first, last, '{' >> int_[std::cout << _1 << '
'] >> '}');
```

There are more ways to bind semantic action functions, but the examples above are the most common. Attaching semantic actions is the first hurdle one has to tackle when getting started with parsing with Spirit. Familiarize yourself with this task and get intimate with the tools behind it such as Boost.Bind and Boost.Lambda.

The examples above can be found here: `../../example/qi/actions.cpp`

**Phoenix**

Phoenix, a companion library bundled with Spirit, is specifically suited for binding semantic actions. It is like Boost.Lambda on steroids, with special custom features that make it easy to integrate semantic actions with Spirit. If your requirements go beyond simple to moderate parsing, it is suggested that you use this library. All the following examples in this tutorial will use Phoenix for semantic actions.

---

**Important**

There are different ways to write semantic actions for Spirit.Qi: using plain functions, Boost.Bind, Boost.Lambda, or Phoenix. The latter three allow you to use special placeholders to control parameter placement (_1, _2, etc.). Each of those libraries has its own implementation of the placeholders, all in different namespaces. You have to make sure not to mix placeholders with a library they don’t belong to and not to use different libraries while writing a semantic action.

Generally, for Boost.Bind, use : : _1, : : _2, etc. (yes, these placeholders are defined in the global namespace).

For Boost.Lambda use the placeholders defined in the namespace boost::lambda.

For semantic actions written using Phoenix use the placeholders defined in the namespace boost::spirit. Please note that all existing placeholders for your convenience are also available from the namespace boost::spirit::qi.

---

**Complex - Our first complex parser**

Well, not really a complex parser, but a parser that parses complex numbers. This time, we're using Phoenix to do the semantic actions.

Here's a simple parser expression for complex numbers:

```cpp
'(' >> double_ >> -(' >> double_) >> ')
| double_
```

What's new? Well, we have:

1. Alternates: e.g. a | b. Try a first. If it succeeds, good. If not, try the next alternative, b.
2. Optionals: e.g. -p. Match the parser p zero or one time.

The complex parser presented above reads as:

- One or two real numbers in parentheses, separated by comma (the second number is optional)
- OR a single real number.

This parser can parse complex numbers of the form:
Here goes, this time with actions:

```cpp
namespace client {
  template <typename Iterator>
  bool parse_complex(Iterator first, Iterator last, std::complex<double>& c)
  {
    using boost::spirit::qi::double_
    using boost::spirit::qi::_1;
    using boost::spirit::qi::phrase_parse;
    using boost::spirit::ascii::space;
    using boost::phoenix::ref;
    double rN = 0.0;
    double iN = 0.0;
    bool r = phrase_parse(first, last,
      // Begin grammar
      ("(' >> double_\[\[ref(rN) = _1\]\]
      >> -\{",' >> double_\[\[ref(iN) = _1\]\]\] >> ")' nN
      | double_\[\[ref(rN) = _1\]\] rN)
      ,
      // End grammar
      space);
    if (!r || first != last) // fail if we did not get a full match
      return false;
    c = std::complex<double>(rN, iN);
    return r;
  }
}
```

The full cpp file for this example can be found here: ../../example/qi/complex_number.cpp

**Note**

Those with experience using *Phoenix* might be confused with the placeholders that we are using (i.e. _1, _2, etc.). Please be aware that we are not using the same placeholders supplied by Phoenix. Take note that we are pulling in the placeholders from namespace `boost::spirit::qi`. These placeholders are specifically tailored for Spirit.

The `double_` parser attaches this action:

```
ref(n) = _1
```

This assigns the parsed result (actually, the attribute of `double_`) to `n`. `ref(n)` tells Phoenix that `n` is a mutable reference. `_1` is a Phoenix placeholder for the parsed result attribute.

**Sum - adding numbers**

Here's a parser that sums a comma-separated list of numbers.

Ok we've glossed over some details in our previous examples. First, our includes:
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <iostream>
#include <string>

Then some using directives:

```cpp
namespace qi = boost::spirit::qi;
namespace ascii = boost::spirit::ascii;
namespace phoenix = boost::phoenix;

using qi::double_;  
using qi::_1;
using ascii::space;
using phoenix::ref;
```

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::phoenix</td>
<td>All of phoenix</td>
</tr>
<tr>
<td>boost::spirit</td>
<td>All of spirit</td>
</tr>
<tr>
<td>boost::spirit::qi</td>
<td>All of spirit.qi</td>
</tr>
<tr>
<td>boost::spirit::ascii</td>
<td>ASCII version of char_ and all char related parsers. Other encodings are also provided (e.g. also an ISO8859.1)</td>
</tr>
<tr>
<td>boost::spirit::arg_names</td>
<td>Special phoenix placeholders for spirit</td>
</tr>
</tbody>
</table>

**Note**

If you feel uneasy with using whole namespaces, feel free to qualify your code, use namespace aliases, etc. For the purpose of this tutorial, we will be presenting unqualified names for both Spirit and Phoenix. No worries, we will always present the full working code, so you won’t get lost. In fact, all examples in this tutorial have a corresponding cpp file that QuickBook (the documentation tool we are using) imports in here as code snippets.

Now the actual parser:
template <typename Iterator>
bool adder(Iterator first, Iterator last, double& n) {
    bool r = qi::phrase_parse(first, last,

        // Begin grammar
        (double_[ref(n) = _1] >> *(',' >> double_[ref(n) += _1]))
    ,

        // End grammar
        space);

    if (first != last) // fail if we did not get a full match
        return false;

    return r;
}

The full cpp file for this example can be found here: ../../example/qi/sum.cpp

This is almost like our original numbers list example. We're incrementally building on top of our examples. This time though, like in the complex number example, we'll be adding the smarts. There's an accumulator (double& n) that adds the numbers parsed. On a successful parse, this number is the sum of all the parsed numbers.

The first double_ parser attaches this action:

    ref(n) = _1

This assigns the parsed result (actually, the attribute of double_) to n. ref(n) tells Phoenix that n is a mutable reference. _1 is a Phoenix placeholder for the parsed result attribute.

The second double_ parser attaches this action:

    ref(n) += _1

So, subsequent numbers add into n.

That wasn't too bad, was it :-) ?

**Number List - stuffing numbers into a std::vector**

This sample demonstrates a parser for a comma separated list of numbers. The numbers are inserted in a vector using phoenix.
template <typename Iterator>
bool parse_numbers(Iterator first, Iterator last, std::vector<double>& v)
{
    using qi::double_;  //  Begin grammar
    using qi::phrase_parse;
    using qi::_1;
    using ascii::space;
    using phoenix::push_back;

    bool r = phrase_parse(first, last,
        // Begin grammar
        (double_[push_back(phoenix::ref(v), _1)]
         >> *(',' >> double_[push_back(phoenix::ref(v), _1)]))
        ,
        // End grammar
        space);

    if (first != last) // fail if we did not get a full match
        return false;
    return r;
}

The full cpp file for this example can be found here: ../../example/qi/num_list2.cpp

This, again, is the same parser as before. This time, instead of summing up the numbers, we stuff them in a std::vector. push_back is supplied by Phoenix. The expression:

push_back(ref(v), _1)

appends the parsed number. Like before, _1 is a Phoenix placeholder for the parsed result attribute. Also, like before, ref(v) tells Phoenix that v, the std::vector, is a mutable reference.

Number List Redux - list syntax

So far, we've been using the syntax:

double_ >> *(',' >> double_)

to parse a comma-delimited list of numbers. Such lists are common in parsing and Spirit provides a simpler shortcut for them. The expression above can be simplified to:

double_ % ','

read as: a list of doubles separated by ','.

This sample, again a variation of our previous example, demonstrates just that:
template <typename Iterator>
bool parse_numbers(Iterator first, Iterator last, std::vector<double>& v)
{
    using qi::double_;
    using qi::phrase_parse;
    using qi::_1;
    using ascii::space;
    using phoenix::push_back;

    bool r = phrase_parse(first, last,
                          // Begin grammar
                          (double_[push_back(phoenix::ref(v), _1)] % ',')
                          ,
                          // End grammar
                          space);

    if (first != last) // fail if we did not get a full match
        return false;
    return r;
}

The full cpp file for this example can be found here: ../../example/qi/num_list3.cpp

Number List Attribute - one more, with style

You've seen that the double_ parser has a double attribute. All parsers have an attribute, even complex parsers. Those that are composed from primitives using operators, like the list parser, also have an attribute. It so happens that the attribute of a list parser:

\[
\text{p \% d}
\]

is a std::vector of the attribute of p. So, for our parser:

\[
\text{double_ \% ','}
\]

we'll have an attribute of:

\[
\text{std::vector<double>}
\]

So, what does this give us? Well, we can simply pass in a std::vector<double> to our number list parser and it will happily churn out our result in our vector. For that to happen, we'll use a variation of the phrase_parse with an additional argument: the parser's attribute. With the following arguments passed to phrase_parse

1. An iterator pointing to the start of the input
2. An iterator pointing to one past the end of the input
3. The parser object
4. Another parser called the skip parser
5. The parser's attribute

our parser now is further simplified to:
template<typename Iterator>
bool parse_numbers(Iterator first, Iterator last, std::vector<double>& v)
{
    using qi::double_;  
    using qi::phrase_parse;  
    using qi::_1;  
    using ascii::space;  

    bool r = phrase_parse(first, last,
    // Begin grammar
    (double_ % ','),
    // End grammar
    space, v);

    if (first != last) // fail if we did not get a full match
        return false;
    return r;
}

The full cpp file for this example can be found here: ../../example/qi/num_list4.cpp

Hey, no more actions!!! Now we're entering the realm of attribute grammars. Cool eh?

**Roman Numerals**

This example demonstrates:

- symbol table
- rule
- grammar

**Symbol Table**

The symbol table holds a dictionary of symbols where each symbol is a sequence of characters (char, wchar_t, int, enumeration etc.). The template class, parameterized by the character type, can work efficiently with 8, 16, 32 and even 64 bit characters. Mutable data of type T are associated with each symbol.

Traditionally, symbol table management is maintained separately outside the BNF grammar through semantic actions. Contrary to standard practice, the Spirit symbol table class **symbols** is a parser. An object of which may be used anywhere in the EBNF grammar specification. It is an example of a dynamic parser. A dynamic parser is characterized by its ability to modify its behavior at run time. Initially, an empty symbols object matches nothing. At any time, symbols may be added or removed, thus, dynamically altering its behavior.

Each entry in a symbol table has an associated mutable data slot. In this regard, one can view the symbol table as an associative container (or map) of key-value pairs where the keys are strings.

The symbols class expects two template parameters. The first parameter specifies the character type of the symbols. The second specifies the data type associated with each symbol: its attribute.

Here's a parser for roman hundreds (100..900) using the symbol table. Keep in mind that the data associated with each slot is the parser's attribute (which is passed to attached semantic actions).
Here's a parser for roman tens (10..90):

```cpp
struct tens : qi::symbols<char, unsigned>
{
    tens()
    {
        add
        (*"X" , 10)
        (*"XX" , 20)
        (*"XXX" , 30)
        (*"XL" , 40)
        (*"L" , 50)
        (*"LX" , 60)
        (*"LXX" , 70)
        (*"LXXX" , 80)
        (*"XC" , 90)
    }
} tens;
```

and, finally, for ones (1..9):
struct ones_ : qi::symbols<char, unsigned>
{
    ones_()
    {
        add
        ("I" , 1)
        ("II" , 2)
        ("III" , 3)
        ("IV" , 4)
        ("V" , 5)
        ("VI" , 6)
        ("VII" , 7)
        ("VIII" , 8)
        ("IX" , 9)
    }
}
ones;

Now we can use hundreds, tens and ones anywhere in our parser expressions. They are all parsers.

Rules

Up until now, we've been inlining our parser expressions, passing them directly to the phrase_parse function. The expression evaluates into a temporary, unnamed parser which is passed into the phrase_parse function, used, and then destroyed. This is fine for small parsers. When the expressions get complicated, you'd want to break the expressions into smaller easier-to-understand pieces, name them, and refer to them from other parser expressions by name.

A parser expression can be assigned to what is called a "rule". There are various ways to declare rules. The simplest form is:

```cpp
rule<Iterator> r;
```

At the very least, the rule needs to know the iterator type it will be working on. This rule cannot be used with phrase_parse. It can only be used with the parse function -- a version that does not do white space skipping (does not have the skipper argument). If you want to have it skip white spaces, you need to pass in the type skip parser, as in the next form:

```cpp
rule<Iterator, Skipper> r;
```

Example:

```cpp
rule<std::string::iterator, space_type> r;
```

This type of rule can be used for both phrase_parse and parse.

For our next example, there's one more rule form you should know about:

```cpp
rule<Iterator, Signature> r;
```

or

```cpp
rule<Iterator, Signature, Skipper> r;
```
rule<Iterator, Signature, Skipper> r;

**Tip**

All rule template arguments after Iterator can be supplied in any order.

The Signature specifies the attributes of the rule. You've seen that our parsers can have an attribute. Recall that the `double_` parser has an attribute of `double`. To be precise, these are *synthesized* attributes. The parser "synthesizes" the attribute value. Think of them as function return values.

There's another type of attribute called "inherited" attribute. We won't need them for now, but it's good that you be aware of such attributes. You can think of them as function arguments. And, rightly so, the rule signature is a function signature of the form:

```cpp
result(argN, argN,..., argN)
```

After having declared a rule, you can now assign any parser expression to it. Example:

```cpp
r = double_ >> *(',' >> double_);
```

**Grammars**

A grammar encapsulates one or more rules. It has the same template parameters as the rule. You declare a grammar by:

1. deriving a struct (or class) from the `grammar` class template
2. declare one or more rules as member variables
3. initialize the base grammar class by giving it the start rule (its the first rule that gets called when the grammar starts parsing)
4. initialize your rules in your constructor

The roman numeral grammar is a very nice and simple example of a grammar:
template <typename Iterator>
struct roman : qi::grammar<Iterator, unsigned()>
{
    roman() : roman::base_type(start)
    {
        using qi::eps;
        using qi::lit;
        using qi::_val;
        using qi::_1;
        using ascii::char_;

        start = eps [_val = 0] >>
        {
            +lit('M') [_val += 1000]
            | hundreds [_val += _1]
            | tens [_val += _1]
            | ones [_val += _1]
        }
    }

    qi::rule<Iterator, unsigned()> start;
};

Things to take notice of:

- The grammar and start rule signature is unsigned(). It has a synthesized attribute (return value) of type unsigned with no inherited attributes (arguments).

- We did not specify a skip-parser. We don't want to skip in between the numerals.

- roman::base_type is a typedef for grammar<Iterator, unsigned()> . If roman was not a template, you could simply write: base_type(start)

- It's best to make your grammar templates such that they can be reused for different iterator types.

- _val is another Phoenix placeholder representing the rule's synthesized attribute.

- eps is a special spirit parser that consumes no input but is always successful. We use it to initialize _val, the rule's synthesized attribute, to zero before anything else. The actual parser starts at +char_('M'), parsing roman thousands. Using eps this way is good for doing pre and post initializations.

- The expression a || b reads: match a or b and in sequence. That is, if both a and b match, it must be in sequence; this is equivalent to a >> -b | b, but more efficient.
Let's Parse!

```cpp
bool r = parse(iter, end, roman_parser, result);

if (r && iter == end)
{
    std::cout << "-------------------------\n";
    std::cout << "Parsing succeeded\n";
    std::cout << "result = " << result << std::endl;
    std::cout << "-------------------------\n";
}
else
{
    std::string rest(iter, end);
    std::cout << "-------------------------\n";
    std::cout << "Parsing failed\n";
    std::cout << "stopped at: ": " << rest << \"\n\";
    std::cout << "-------------------------\n";
}
```

roman_parser is an object of type roman, our roman numeral parser. This time around we are using the no-skipping version of the parse functions. We do not want to skip any spaces! We are also passing in an attribute, unsigned result, which will receive the parsed value.

The full cpp file for this example can be found here: ../../example/qi/roman.cpp

### Employee - Parsing into structs

It's a common question in the Spirit General List: How do I parse and place the results into a C++ struct? Of course, at this point, you already know various ways to do it, using semantic actions. There are many ways to skin a cat. Spirit2, being fully attributed, makes it even easier. The next example demonstrates some features of Spirit2 that make this easy. In the process, you'll learn about:

- More about attributes
- Auto rules
- Some more built-in parsers
- Directives

First, let's create a struct representing an employee:

```cpp
struct employee
{
    int age;
    std::string surname;
    std::string forename;
    double salary;
};
```

Then, we need to tell Boost.Fusion about our employee struct to make it a first-class fusion citizen that the grammar can utilize. If you don't know fusion yet, it is a Boost library for working with heterogeneous collections of data, commonly referred to as tuples. Spirit uses fusion extensively as part of its infrastructure.

In fusion's view, a struct is just a form of a tuple. You can adapt any struct to be a fully conforming fusion tuple:
BOOST_FUSION_ADAPT_STRUCT(
    client::employee,
    (int, age)
    (std::string, surname)
    (std::string, forename)
    (double, salary)
)

Now we'll write a parser for our employee. Inputs will be of the form:

    employee{ age, "surname", "forename", salary }

Here goes:

    template <typename Iterator>
    struct employee_parser : qi::grammar<Iterator, employee(), ascii::space_type>
    {
        employee_parser() : employee_parser::base_type(start)
        {
            using qi::int_;  // integer literal
            using qi::lit;  // literal string
            using qi::double_;  // double literal
            using qi::lexeme;  // any lexeme
            using ascii::char_;  // any character

            quoted_string %= lexeme['"' >> +(char_ - '"') >> '"'];

            start %=
                lit("employee")
                >> '{'
                >> int_ >> ','
                >> quoted_string >> ','
                >> quoted_string >> ','
                >> double_ >> '}'
                ;
        }

        qi::rule<Iterator, std::string(), ascii::space_type> quoted_string;
        qi::rule<Iterator, employee(), ascii::space_type> start;
    };

The full cpp file for this example can be found here: ../../example/qi/employee.cpp

Let's walk through this one step at a time (not necessarily from top to bottom).

    template <typename Iterator>
    struct employee_parser : grammar<Iterator, employee(), space_type>

employee_parser is a grammar. Like before, we make it a template so that we can reuse it for different iterator types. The grammar's signature is:

    employee()

meaning, the parser generates employee structs. employee_parser skips white spaces using space_type as its skip parser.
employee_parser() : employee_parser::base_type(start)

Initializes the base class.

```cpp
rule<Iterator, std::string(), space_type> quoted_string;
rule<Iterator, employee(), space_type> start;
```

Declares two rules: `quoted_string` and `start`. `start` has the same template parameters as the grammar itself. `quoted_string` has a `std::string` attribute.

**Lexeme**

```cpp
lexeme["" >> +(char_ - '"") >> '"'];
```

`lexeme` inhibits space skipping from the open brace to the closing brace. The expression parses quoted strings.

```cpp
+(char_ - '"")
```

parses one or more chars, except the double quote. It stops when it sees a double quote.

**Difference**

The expression:

```cpp
a - b
```
parses `a` but not `b`. Its attribute is just `A`; the attribute of `a`. `b`'s attribute is ignored. Hence, the attribute of:

```cpp
char_ - '"'
```
is just `char`.

**Plus**

```cpp
+a
```
is similar to Kleene star. Rather than match everything, `+a` matches one or more. Like its related function, the Kleene star, its attribute is a `std::vector<A>` where `A` is the attribute of `a`. So, putting all these together, the attribute of

```cpp
+(char_ - '"")
```
is then:

```cpp
std::vector<char>
```

**Sequence Attribute**

Now what’s the attribute of

```cpp
"" >> +(char_ - '"') >> "";
```

?
Well, typically, the attribute of:

```
a >> b >> c
```

is:

```
fusion::vector<A, B, C>
```

where \( A \) is the attribute of \( a \), \( B \) is the attribute of \( b \) and \( C \) is the attribute of \( c \). What is \( fusion::vector? \) - a tuple.

**Note**

If you don’t know what I am talking about, see: **Fusion Vector.** It might be a good idea to have a look into **Boost.Fusion** at this point. You’ll definitely see more of it in the coming pages.

**Attribute Collapsing**

Some parsers, especially those very little literal parsers you see, like ‘”’, do not have attributes.

Nodes without attributes are disregarded. In a sequence, like above, all nodes with no attributes are filtered out of the \( fusion::vector \). So, since ‘”’ has no attribute, and \(+\text{(char}_-\text{'”')}\) has a \( \text{std::vector<char>} \) attribute, the whole expression’s attribute should have been:

```
fusion::vector<\text{std::vector<char>}> 
```

But wait, there’s one more collapsing rule: If the attribute is followed by a single element \( fusion::vector \), The element is stripped naked from its container. To make a long story short, the attribute of the expression:

```
'"" >> +\text{(char}_-\text{'"')} >> '"
```

is:

```
\text{std::vector<char>}
```

**Auto Rules**

It is typical to see rules like:

```
r = p[_val = _1]; 
```

If you have a rule definition such as the above, where the attribute of the RHS (right hand side) of the rule is compatible with the attribute of the LHS (left hand side), then you can rewrite it as:

```
r %= p; 
```

The attribute of \( p \) automatically uses the attribute of \( r \).

So, going back to our quoted_string rule:

```
quoted_string %= lexeme["" >> +(char_ - "") >> "]
```

is a simplified version of:
The attribute of the quoted_string rule: std::string is compatible with the attribute of the RHS: std::vector<char>. The RHS extracts the parsed attribute directly into the rule's attribute, in-situ.

**Note**

r %= p and r = p are equivalent if there are no semantic actions associated with p.

**Finally**

We're down to one rule, the start rule:

```
start %=
  lit("employee")
  >> '{'
  >> int_ >> '','
  >> quoted_string >> ',,'
  >> quoted_string >> ',,'
  >> double_
  >> '}
;
```

Applying our collapsing rules above, the RHS has an attribute of:

```
fusion::vector<int, std::string, std::string, double>
```

These nodes do not have an attribute:

- lit("employee")
- '
- ','
- ',
- '}'

**Note**

In case you are wondering, lit("employee") is the same as "employee". We had to wrap it inside `lit` because immediately after it is >> '{'. You can’t right-shift a char[] and a char - you know, C++ syntax rules.

Recall that the attribute of start is the employee struct:

```
struct employee
{
  int age;
  std::string surname;
  std::string forename;
  double salary;
};
```

Now everything is clear, right? The struct employee IS compatible with fusion::vector<int, std::string, std::string, double>. So, the RHS of start uses start's attribute (a struct employee) in-situ when it does its work.
Mini XML - ASTs!

Stop and think about it... We've come very close to generating an AST (abstract syntax tree) in our last example. We parsed a single structure and generated an in-memory representation of it in the form of a struct: the struct employee. If we changed the implementation to parse one or more employees, the result would be a std::vector<employee>. We can go on and add more hierarchy: teams, departments, corporations. Then we'll have an AST representation of it all.

In this example (actually two examples), we'll now explore how to create ASTs. We will parse a minimalistic XML-like language and compile the results into our data structures in the form of a tree.

Along the way, we'll see new features:

- Inherited attributes
- Variant attributes
- Local Variables
- Not Predicate
- Lazy Lit

The full cpp files for these examples can be found here: ../../example/qi/mini_xml1.cpp and here: ../../example/qi/mini_xml2.cpp

There are a couple of sample toy-xml files in the mini_xml_samples subdirectory: ../../example/qi/mini_xml_samples/1.toyxml, ../../example/qi/mini_xml_samples/2.toyxml, and ../../example/qi/mini_xml_samples/3.toyxml for testing purposes. The example ../../example/qi/mini_xml_samples/4.toyxml has an error in it.

First Cut

Without further delay, here's the first version of the XML grammar:
template <typename Iterator>
struct mini_xml_grammar : qi::grammar<Iterator, mini_xml(), ascii::space_type>
{
    mini_xml_grammar() : mini_xml_grammar::base_type(xml)
    {
using qi::lit;
using qi::lexeme;
using ascii::char_
using ascii::string;
using namespace qi::labels;
using phoenix::at_c;
using phoenix::push_back;

text = lexeme[(char_ - '<') [_val += _1]]; // text rule
[node = (xml | text) [_val = _1]]
start_tag =
    '<'
    >> !lit('/')
    >> lexeme[(char_ - '>') [_val += _1]]
    >> '>
;
end_tag =
    '</'
    >> string(_r1)
    >> '>
;
xml =
    start_tag
    >> node
    >> end_tag(at_c<0>(_val))
;
}
qi::rule<Iterator, mini_xml(), ascii::space_type> xml;
qi::rule<Iterator, mini_xml_node(), ascii::space_type> node;
qi::rule<Iterator, std::string(), ascii::space_type> text;
qi::rule<Iterator, std::string(), ascii::space_type> start_tag;
qi::rule<Iterator, void(std::string), ascii::space_type> end_tag;
};

Going bottom up, let's examine the text rule:

```
rule<Iterator, std::string(), space_type> text;
```

and its definition:

```
text = lexeme[(char_ - '<') [_val += _1]];
```

The semantic action collects the chars and appends them (via +=) to the std::string attribute of the rule (represented by the placeholder _val).

**Alternates**

```
rule<Iterator, mini_xml_node(), space_type> node;
```

and its definition:
node = (xml | text) [_val = _1];

We'll see a `mini_xml_node` structure later. Looking at the rule definition, we see some alternation going on here. An `xml node` is either an `xml` OR `text`. Hmmm... hold on to that thought...

```cpp
rule<Iterator, std::string(), space_type> start_tag;
```

Again, with an attribute of `std::string`. Then, it's definition:

```cpp
start_tag =
  '<'
  >> !char_('/')
  >> lexeme[+(char_ - '>')] [_val += _1]
  >> '>
;
```

**Not Predicate**

`start_tag` is similar to the `text` rule apart from the added `'<'` and `'>'`. But wait, to make sure that the `start_tag` does not parse `end_tags` too, we add: `!char_('/')`. This is a "Not Predicate":

```cpp
!p
```

It will try the parser, `p`. If it is successful, fail; otherwise, pass. In other words, it negates the result of `p`. Like the `eps`, it does not consume any input though. It will always rewind the iterator position to where it was upon entry. So, the expression:

```cpp
!char_('/')
```

basically says: we should not have a `'/'` at this point.

**Inherited Attribute**

The `end_tag`:

```cpp
rule<Iterator, void(std::string), space_type> end_tag;
```

Ohh! Now we see an inherited attribute there: `std::string`. The `end_tag` does not have a synthesized attribute. Let's see its definition:

```cpp
end_tag =
  "</"
  >> lit(_r1)
  >> '>
;
```

_r1_ is yet another `Phoenix` placeholder for the first inherited attribute (we have only one, use _r2, _r3, etc. if you have more).

**A Lazy Lit**

Check out how we used `lit` here, this time, not with a literal string, but with the value of the first inherited attribute, which is specified as `std::string` in our rule declaration.

Finally, our `xml` rule:
mini_xml is our attribute here. We’ll see later what it is. Let’s see its definition:

```cpp
xml =
    start_tag [at_c<0>(_val) = _1]
    >> *node [push_back(at_c<1>(_val), _1)]
    >> end_tag(at_c<0>(_val));
```

Those who know Boost.Fusion now will notice `at_c<0>` and `at_c<1>`. This gives us a hint that mini_xml is a sort of a tuple - a fusion sequence. `at_c<N>` here is a lazy version of the tuple accessors, provided by Phoenix.

### How it all works

So, what’s happening?

1. Upon parsing `start_tag`, the parsed start-tag string is placed in `at_c<0>(_val)`.

2. Then we parse zero or more nodes. At each step, we `push_back` the result into `at_c<1>(_val)`.

3. Finally, we parse the `end_tag` giving it an inherited attribute: `at_c<0>(_val)`. This is the string we obtained from the `start_tag`. Investigate `end_tag` above. It will fail to parse if it gets something different from what we got from the `start_tag`. This ensures that our tags are balanced.

To give the last item some more light, what happens is this:

```cpp
end_tag(at_c<0>(_val))
```

calls:

```cpp
end_tag =
    "</" [lit(_r1)] [">"]
```

passing in `at_c<0>(_val)`, the string from start tag. This is referred to in the `end_tag` body as `_r1`.

### The Structures

Let’s see our structures. It will definitely be hierarchical: xml is hierarchical. It will also be recursive: xml is recursive.


```cpp
struct mini_xml;

typedef
    boost::variant<
        boost::recursive_wrapper<mini_xml>,
        std::string
    > mini_xml_node;

struct mini_xml
{
    std::string name; // tag name
    std::vector<mini_xml_node> children; // children
};
```

Of Alternates and Variants

So that's what a `mini_xml_node` looks like. We had a hint that it is either a `string` or a `mini_xml`. For this, we use `Boost.Variant`. `boost::recursive_wrapper` wraps `mini_xml`, making it a recursive data structure.

Yep, you got that right: the attribute of an alternate:

```
a | b
```

is a

```
boost::variant<A, B>
```

where A is the attribute of a and B is the attribute of b.

Adapting structs again

`mini_xml` is no brainier. It is a plain ol' struct. But as we've seen in our employee example, we can adapt that to be a `Boost.Fusion` sequence:

```
BOOST_FUSION_ADAPT_STRUCT(
    client::mini_xml,
    (std::string, name)
    (std::vector<client::mini_xml_node>, children)
)
```

One More Take

Here's another version. The AST structure remains the same, but this time, you'll see that we make use of auto-rules making the grammar semantic-action-less. Here it is:
template <typename Iterator>
struct mini_xml_grammar
  : qi::grammar<Iterator, mini_xml(), qi::locals<std::string>, ascii::space_type>
{

  mini_xml_grammar()
  : mini_xml_grammar::base_type(xml)
  {
    using qi::lit;
    using qi::lexeme;
    using ascii::char_;
    using ascii::string;
    using namespace qi::labels;

    text %= lexeme[+(char_ - '<')];
    node %= xml | text;

    start_tag %=
      '<'
      >> !lit('/')
      >> lexeme[+(char_ - '>')]
      >> '>
    ;

    end_tag =
      "</"
      >> string(_r1)
      >> '>
    ;

    xml %=
      start_tag[_a = _1]
      >> *node
      >> end_tag(_a)
    ;
  }

qi::rule<Iterator, mini_xml(), qi::locals<std::string>, ascii::space_type> xml;
qi::rule<Iterator, mini_xml_node(), ascii::space_type> node;
qi::rule<Iterator, std::string(), ascii::space_type> text;
qi::rule<Iterator, std::string(), ascii::space_type> start_tag;
qi::rule<Iterator, void(std::string), ascii::space_type> end_tag;
};

This one shouldn’t be any more difficult to understand after going through the first xml parser example. The rules are almost the same, except that, we got rid of semantic actions and used auto-rules (see the employee example if you missed that). There is some new stuff though. It’s all in the xml rule:

**Local Variables**

```
rule<Iterator, mini_xml(), locals<std::string>, space_type> xml;
```

Wow, we have four template parameters now. What’s that **locals** guy doing there? Well, it declares that the rule xml will have one local variable: a **string**. Let’s see how this is used in action:
Upon parsing `start_tag`, the parsed start-tag string is placed in the local variable specified by (yet another) Phoenix placeholder: `_a`. We have only one local variable. If we had more, these are designated by `_b.._z`.

2. Then we parse zero or more nodes.

3. Finally, we parse the `end_tag` giving it an inherited attribute: `_a`, our local variable.

There are no actions involved in stuffing data into our `xml` attribute. It’s all taken care of thanks to the auto-rule.

### Mini XML - Error Handling

A parser will not be complete without error handling. Spirit2 provides some facilities to make it easy to adapt a grammar for error handling. We’ll wrap up the Qi tutorial with another version of the mini xml parser, this time, with error handling.

The full cpp file for this example can be found here: `./example/qi/mini_xml3.cpp`

Here’s the grammar: 

```xml
xml %=
  start_tag[_a = 1]
  *node
  end_tag[_a]
;```
template <typename Iterator>
struct mini_xml_grammar
 : qi::grammar<Iterator, mini_xml(), qi::locals<std::string>, ascii::space_type>
{
    mini_xml_grammar()
    : mini_xml_grammar::base_type(xml, "xml")
    {
        using qi::lit;
        using qi::lexeme;
        using qi::on_error;
        using qi::fail;
        using ascii::char_;  
        using ascii::string;
        using namespace qi::labels;

        using phoenix::construct;
        using phoenix::val;

        text %= lexeme[+(char_ - '<')];
        node %= xml | text;

        start_tag %=
            '<'  
            >> !lit('/')  
            > lexeme[+(char_ - '>')]  
            > '>'  
            ;

        end_tag =  
            "</"  
            > string(_r1)  
            > '>'  
            ;

        xml %=
            start_tag[_a = _1]  
            > *node  
            > end_tag(_a)  
            ;

        xml.name("xml");
        node.name("node");
        text.name("text");
        start_tag.name("start_tag");
        end_tag.name("end_tag");

        on_error<fail>
        {
            xml
            , std::cout
            << val("Error! Expecting ")
            << _4  
            // what failed?
            << val(" here: ")
            << construct<std::string>(_3, _2)  
            // iterators to error-pos, end
            << val(""")
            << std::endl
            ;
        }
    }
}
What's new?

Readable Names

First, when we call the base class, we give the grammar a name:

```cpp
: mini_xml_grammar::base_type(xml, "xml")
```

Then, we name all our rules:

```cpp
xml.name("xml");
node.name("node");
text.name("text");
start_tag.name("start_tag");
end_tag.name("end_tag");
```

On Error

`on_error` declares our error handler:

```cpp
on_error<Action>(rule, handler)
```

This will specify what we will do when we get an error. We will print out an error message using phoenix:

```cpp
on_error<fail>(
    xml,
    std::cout
    << val("Error! Expecting ")
    << _4  // what failed?
    << val(" here: \\
''
    << construct<std::string>(_3, _2)  // iterators to error-pos, end
    << val("\")
    << std::endl
);
```

we choose to `fail` in our example for the `Action`: Quit and fail. Return a `no_match` (false). It can be one of:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fail</td>
<td>Quit and fail. Return a <code>no_match</code>.</td>
</tr>
<tr>
<td>retry</td>
<td>Attempt error recovery, possibly moving the iterator position.</td>
</tr>
<tr>
<td>accept</td>
<td>Force error recovery, possibly moving the iterator position appropriately.</td>
</tr>
<tr>
<td>rethrow</td>
<td>Rethrows the error.</td>
</tr>
</tbody>
</table>

`rule` is the rule to which the handler is attached. In our case, we are attaching to the `xml` rule.
handler is the actual error handling function. It expects 4 arguments:

<table>
<thead>
<tr>
<th>Arg</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>The position of the iterator when the rule with the handler was entered.</td>
</tr>
<tr>
<td>last</td>
<td>The end of input.</td>
</tr>
<tr>
<td>error-pos</td>
<td>The actual position of the iterator where the error occurred.</td>
</tr>
<tr>
<td>what</td>
<td>What failed: a string describing the failure.</td>
</tr>
</tbody>
</table>

**Expectation Points**

You might not have noticed it, but some of our expressions changed from using the `>>` to `>`. Look, for example:

```cpp
end_tag = 
  "</" 
  > lit(_r1)   
  > '>'
;```

What is it? It’s the expectation operator. You will have some "deterministic points" in the grammar. Those are the places where backtracking cannot occur. For our example above, when you get a "</", you definitely must see a valid end-tag label next. It should be the one you got from the start-tag. After that, you definitely must have a '>' next. Otherwise, there is no point in proceeding and trying other branches, regardless where they are. The input is definitely erroneous. When this happens, an expectation_failure exception is thrown. Somewhere outward, the error handler will catch the exception.

Try building the parser: `../../example/qi/mini_xml3.cpp`. You can find some examples in: `../../example/qi/mini_xml_samples` for testing purposes. "4.toyxml" has an error in it:

```xml
<foo><bar></foo></bar>
```

Running the example with this gives you:

```
Error! Expecting "bar" here: "foo"></bar"
Error! Expecting end_tag here: "<bar></foo></bar"
-------------------------
Parsing failed
-------------------------
```

**Quick Reference**

This quick reference section is provided for convenience. You can use this section as a sort of a "cheat-sheet" on the most commonly used Qi components. It is not intended to be complete, but should give you an easy way to recall a particular component without having to dig up on pages and pages of reference documentation.

**Common Notation**

**Notation**

<table>
<thead>
<tr>
<th>P</th>
<th>Parser type</th>
</tr>
</thead>
<tbody>
<tr>
<td>p, a, b, c</td>
<td>Parser objects</td>
</tr>
</tbody>
</table>
A, B, C
Attribute types of parsers a, b and c

I
The iterator type used for parsing

Unused
An unused_type

Context
The enclosing rule’s Context type

attrib
An attribute value

Attrib
An attribute type

b
A boolean expression

fp
A (lazy parser) function with signature P(Unused, Context)

fa
A (semantic action) function with signature void(Attrib, Context, bool&). The third parameter is a boolean flag that can be set to false to force the parse to fail. Both Context and the boolean flag are optional.

first
An iterator pointing to the start of input

last
An iterator pointing to the end of input

Ch
Character-class specific character type (See Character Class Types)

ch
Character-class specific character (See Character Class Types)

ch2
Character-class specific character (See Character Class Types)

charset
Character-set specifier string (example: "a-z0-9")

str
Character-class specific string (See Character Class Types)

Str
Attribute of str::std::basic_string<T> where T is the underlying character type of str

tuple<> Used as a placeholder for a fusion sequence

vector<> Used as a placeholder for an STL container

variant<> Used as a placeholder for a boost::variant

optional<> Used as a placeholder for a boost::optional

65
Qi Parsers

Character Parsers
<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>Unused</td>
<td>Matches ch</td>
</tr>
<tr>
<td>lit(ch)</td>
<td>Unused</td>
<td>Matches ch</td>
</tr>
<tr>
<td>char_</td>
<td>Ch</td>
<td>Matches any character</td>
</tr>
<tr>
<td>char_(ch)</td>
<td>Ch</td>
<td>Matches ch</td>
</tr>
<tr>
<td>char_(&quot;c&quot;)</td>
<td>Ch</td>
<td>Matches a single char string literal, c</td>
</tr>
<tr>
<td>char_(ch, ch2)</td>
<td>Ch</td>
<td>Matches a range of chars from ch to ch2 (inclusive)</td>
</tr>
<tr>
<td>char_(charset)</td>
<td>Ch</td>
<td>Matches a character set charset</td>
</tr>
<tr>
<td>alnum</td>
<td>Ch</td>
<td>Matches a character based on the equivalent of std::isalnum in the current character set</td>
</tr>
<tr>
<td>alpha</td>
<td>Ch</td>
<td>Matches a character based on the equivalent of std::isalpha in the current character set</td>
</tr>
<tr>
<td>blank</td>
<td>Ch</td>
<td>Matches a character based on the equivalent of std::isblank in the current character set</td>
</tr>
<tr>
<td>cntrl</td>
<td>Ch</td>
<td>Matches a character based on the equivalent of std::iscntrl in the current character set</td>
</tr>
<tr>
<td>digit</td>
<td>Ch</td>
<td>Matches a character based on the equivalent of std::isdigit in the current character set</td>
</tr>
<tr>
<td>graph</td>
<td>Ch</td>
<td>Matches a character based on the equivalent of std::isgraph in the current character set</td>
</tr>
<tr>
<td>print</td>
<td>Ch</td>
<td>Matches a character based on the equivalent of std::isprint in the current character set</td>
</tr>
<tr>
<td>punct</td>
<td>Ch</td>
<td>Matches a character based on the equivalent of std::ispunct in the current character set</td>
</tr>
<tr>
<td>space</td>
<td>Ch</td>
<td>Matches a character based on the equivalent of std::isspace in the current character set</td>
</tr>
<tr>
<td>xdigit</td>
<td>Ch</td>
<td>Matches a character based on the equivalent of std::isxdigit in the current character set</td>
</tr>
<tr>
<td>Expression</td>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>lower</td>
<td>Ch</td>
<td>Matches a character based on the equivalent of <code>std::islower</code> in the current character set</td>
</tr>
<tr>
<td>upper</td>
<td>Ch</td>
<td>Matches a character based on the equivalent of <code>std::isupper</code> in the current character set</td>
</tr>
</tbody>
</table>
Numeric Parsers
<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float_</td>
<td>float</td>
<td>Parse a floating point number into a float</td>
</tr>
<tr>
<td>float_(num)</td>
<td>float</td>
<td>Parse a floating point number into a float, a number is matched only if it's num</td>
</tr>
<tr>
<td>double_</td>
<td>double</td>
<td>Parse a floating point number into a double</td>
</tr>
<tr>
<td>double_</td>
<td>double</td>
<td>Parse a floating point number into a double, a number is matched only if it's num</td>
</tr>
<tr>
<td>long_double</td>
<td>long double</td>
<td>Parse a floating point number into a long double</td>
</tr>
<tr>
<td>long_double</td>
<td>long double</td>
<td>Parse a floating point number into a long double, a number is matched only if it's num</td>
</tr>
<tr>
<td>bin</td>
<td>unsigned</td>
<td>Parse a binary integer into an unsigned</td>
</tr>
<tr>
<td>oct</td>
<td>unsigned</td>
<td>Parse an octal integer into an unsigned</td>
</tr>
<tr>
<td>hex</td>
<td>unsigned</td>
<td>Parse a hexadecimal integer into an unsigned</td>
</tr>
<tr>
<td>ushort_</td>
<td>unsigned short</td>
<td>Parse an unsigned short integer</td>
</tr>
<tr>
<td>ushort_(num)</td>
<td>unsigned short</td>
<td>Parse an unsigned short integer, a number is matched only if it's num</td>
</tr>
<tr>
<td>ulong_</td>
<td>unsigned long</td>
<td>Parse an unsigned long integer</td>
</tr>
<tr>
<td>ulong_(num)</td>
<td>unsigned long</td>
<td>Parse an unsigned long integer, a number is matched only if it's num</td>
</tr>
<tr>
<td>uint_</td>
<td>unsigned int</td>
<td>Parse an unsigned int</td>
</tr>
<tr>
<td>uint_(num)</td>
<td>unsigned int</td>
<td>Parse an unsigned int, a number is matched only if it's num</td>
</tr>
<tr>
<td>ulong_long</td>
<td>unsigned long long</td>
<td>Parse an unsigned long long</td>
</tr>
<tr>
<td>ulong_long_(num)</td>
<td>unsigned long long</td>
<td>Parse an unsigned long long, a number is matched only if it's num</td>
</tr>
<tr>
<td>short_</td>
<td>short</td>
<td>Parse a short integer</td>
</tr>
<tr>
<td>short_(num)</td>
<td>short</td>
<td>Parse a short integer, a number is matched only if it's num</td>
</tr>
<tr>
<td>long_</td>
<td>long</td>
<td>Parse a long integer</td>
</tr>
</tbody>
</table>
### Expression Attribute Description

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>long_(num)</code></td>
<td>long</td>
<td>Parse a long integer, a number is matched only if it's <code>num</code></td>
</tr>
<tr>
<td><code>int_</code></td>
<td>int</td>
<td>Parse an int</td>
</tr>
<tr>
<td><code>int_(num)</code></td>
<td>int</td>
<td>Parse an int, a number is matched only if it's <code>num</code></td>
</tr>
<tr>
<td><code>long_long</code></td>
<td>long long</td>
<td>Parse a long long</td>
</tr>
<tr>
<td><code>long_long(num)</code></td>
<td>long long</td>
<td>Parse a long long, a number is matched only if it's <code>num</code></td>
</tr>
</tbody>
</table>

### String Parsers

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>str</code></td>
<td>Unused</td>
<td>Matches <code>str</code></td>
</tr>
<tr>
<td><code>lit(str)</code></td>
<td>Unused</td>
<td>Matches <code>str</code></td>
</tr>
<tr>
<td><code>string(str)</code></td>
<td>Str</td>
<td>Matches <code>str</code></td>
</tr>
<tr>
<td><code>symbols&lt;&gt;</code></td>
<td>N/A</td>
<td>Declare a symbol table, <code>sym</code>. Ch is the underlying char type of the symbol table keys. T is the data type associated with each key.</td>
</tr>
<tr>
<td>sym.add</td>
<td>N/A</td>
<td>Add symbols into a symbol table, <code>sym</code>. val1 and val2 are optional data of type T, the data type associated with each key.</td>
</tr>
<tr>
<td>sym</td>
<td>T</td>
<td>Matches entries in the symbol table, <code>sym</code>. If successful, returns the data associated with the key.</td>
</tr>
</tbody>
</table>
## Auxiliary Parsers

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eol</code></td>
<td>Unused</td>
<td>Matches the end of line (<code>\r</code> or <code>\n</code> or <code>\r\n</code>)</td>
</tr>
<tr>
<td><code>eoi</code></td>
<td>Unused</td>
<td>Matches the end of input (first <code>==</code> last)</td>
</tr>
<tr>
<td><code>eps</code></td>
<td>Unused</td>
<td>Match an empty string</td>
</tr>
<tr>
<td><code>eps(b)</code></td>
<td>Unused</td>
<td>If <code>b</code> is true, match an empty string</td>
</tr>
<tr>
<td><code>lazy(fp)</code></td>
<td>Attribute of <code>P</code> where <code>P</code> is the return type of <code>fp</code></td>
<td>Invoke <code>fp</code> at parse time, returning a parser <code>p</code> which is then called to parse.</td>
</tr>
<tr>
<td><code>fp</code></td>
<td>see <code>lazy(fp)</code> above</td>
<td>Equivalent to <code>lazy(fp)</code></td>
</tr>
<tr>
<td><code>attr(attrib)</code></td>
<td><code>Attrib</code></td>
<td>Doesn’t consume/parse any input, but exposes the argument <code>attrib</code> as its attribute.</td>
</tr>
</tbody>
</table>

## Binary Parsers

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>byte_</code></td>
<td>8 bits native endian</td>
<td>Matches an 8 bit binary in native endian representation</td>
</tr>
<tr>
<td><code>word</code></td>
<td>16 bits native endian</td>
<td>Matches a 16 bit binary in native endian representation</td>
</tr>
<tr>
<td><code>big_word</code></td>
<td>16 bits big endian</td>
<td>Matches a 16 bit binary in big endian representation</td>
</tr>
<tr>
<td><code>little_word</code></td>
<td>16 bits little endian</td>
<td>Matches a 16 bit binary in little endian representation</td>
</tr>
<tr>
<td><code>dword</code></td>
<td>32 bits native endian</td>
<td>Matches a 32 bit binary in native endian representation</td>
</tr>
<tr>
<td><code>big_dword</code></td>
<td>32 bits big endian</td>
<td>Matches a 32 bit binary in big endian representation</td>
</tr>
<tr>
<td><code>little_dword</code></td>
<td>32 bits little endian</td>
<td>Matches a 32 bit binary in little endian representation</td>
</tr>
<tr>
<td><code>qword</code></td>
<td>64 bits native endian</td>
<td>Matches a 64 bit binary in native endian representation</td>
</tr>
<tr>
<td><code>big_qword</code></td>
<td>64 bits big endian</td>
<td>Matches a 64 bit binary in big endian representation</td>
</tr>
<tr>
<td><code>little_qword</code></td>
<td>64 bits little endian</td>
<td>Matches a 64 bit binary in little endian representation</td>
</tr>
</tbody>
</table>
## Auto Parsers

See here for more information about Auto Parsers.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto_</td>
<td>hold_any</td>
<td>Parse input using a parser created from the supplied attribute type using the <code>create_parser</code> API function.</td>
</tr>
</tbody>
</table>

## Parser Directives

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lexeme[a]</td>
<td>A</td>
<td>Disable skip parsing for a, does pre-skip-ping</td>
</tr>
<tr>
<td>no_skip[a]</td>
<td>A</td>
<td>Disable skip parsing for a, no pre-skip-ping</td>
</tr>
<tr>
<td>no_case[a]</td>
<td>A</td>
<td>Inhibits case-sensitivity for a</td>
</tr>
<tr>
<td>omit[a]</td>
<td>Unused</td>
<td>Ignores the attribute type of a</td>
</tr>
<tr>
<td>matches[a]</td>
<td>bool</td>
<td>Return if the embedded parser a matched its input</td>
</tr>
<tr>
<td>as&lt;T&gt;<a href=""></a>[a]</td>
<td>A</td>
<td>Force atomic assignment for arbitrary attribute types</td>
</tr>
<tr>
<td>as_string[][a]</td>
<td>A</td>
<td>Force atomic assignment for string attributes</td>
</tr>
<tr>
<td>as_wstring[][a]</td>
<td>A</td>
<td>Force atomic assignment for wide character string attributes</td>
</tr>
<tr>
<td>raw[a]</td>
<td><code>boost::iterator_range&lt; I&gt;</code></td>
<td>Presents the transduction of a as an iterator range</td>
</tr>
<tr>
<td>repeat[a]</td>
<td>vector&lt;A&gt;</td>
<td>Repeat a zero or more times</td>
</tr>
<tr>
<td>repeat(N)[a]</td>
<td>vector&lt;A&gt;</td>
<td>Repeat a N times</td>
</tr>
<tr>
<td>repeat(N, M)[a]</td>
<td>vector&lt;A&gt;</td>
<td>Repeat a N to M times</td>
</tr>
<tr>
<td>repeat(N, inf)[a]</td>
<td>vector&lt;A&gt;</td>
<td>Repeat a N or more times</td>
</tr>
<tr>
<td>skip[a]</td>
<td>A</td>
<td>Re-establish the skipper that got inhibited by lexeme or no_skip.</td>
</tr>
<tr>
<td>skip(p)[a]</td>
<td>A</td>
<td>Use p as a skipper for parsing a</td>
</tr>
</tbody>
</table>
### Parser Operators

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!a</td>
<td>Unused</td>
<td>Not predicate. If the predicate a matches, fail. Otherwise, return a zero length match.</td>
</tr>
<tr>
<td>&amp;a</td>
<td>Unused</td>
<td>And predicate. If the predicate a matches, return a zero length match. Otherwise, fail.</td>
</tr>
<tr>
<td>-a</td>
<td>optional&lt;A&gt;</td>
<td>Optional. Parse a zero or one time</td>
</tr>
<tr>
<td>*a</td>
<td>vector&lt;A&gt;</td>
<td>Kleene. Parse a zero or more times</td>
</tr>
<tr>
<td>+a</td>
<td>vector&lt;A&gt;</td>
<td>Plus. Parse a one or more times</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>variant&lt;A, B&gt;</td>
</tr>
<tr>
<td>a &gt;&gt; b</td>
<td>tuple&lt;A, B&gt;</td>
<td>Sequence. Parse a followed by b</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>tuple&lt;A, B&gt;</td>
<td>Expect. Parse a followed by b. b is expected to match when a matches, otherwise, an expectation_failure is thrown.</td>
</tr>
<tr>
<td>a - b</td>
<td>A</td>
<td>Difference. Parse a but not b</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>a % b</td>
<td>vector&lt;A&gt;</td>
<td>List. Parse a delimited b one or more times</td>
</tr>
<tr>
<td>a ^ b</td>
<td>tuple&lt;optional&lt;A&gt;, optional&lt;B&gt;&gt;</td>
<td>Permutation. Parse a or b or a followed by b or b followed by a.</td>
</tr>
</tbody>
</table>

### Parser Semantic Actions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p[fa]</td>
<td>Attribute of p</td>
<td>Call semantic action, fa if p succeeds.</td>
</tr>
</tbody>
</table>

### Compound Attribute Rules

#### Notation

The notation we will use will be of the form:

```
    a: A, b: B, ... --> composite-expression: composite-attribute
```

a, b, etc. are the operands. A, B, etc. are the operand's attribute types. composite-expression is the expression involving the operands and composite-attribute is the resulting attribute type of the composite expression.

For instance:
a: A, b: B --> (a >> b): tuple<A, B>

reads as: given, a and b are parsers, and A is the type of the attribute of a, and B is the type of the attribute of b, then the type of the attribute of a >> b will be tuple<A, B>.

⚠️ Important

In the attribute tables, we will use vector<A> and tuple<A, B...> as placeholders only. The notation of vector<A> stands for any STL container holding elements of type A and the notation tuple<A, B...> stands for any Boost.Fusion sequence holding A, B, ... etc. elements. Finally, Unused stands for unused_type.
Compound Parser Attribute Types
<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
</table>
| **Sequence (a >> b)** | ```
  a: A, b: B --> (a >> b): tuple<A, B>
  a: A, b: Unused --> (a >> b): A
  a: Unused, b: B --> (a >> b): B
  a: Unused, b: Unused --> (a >> b): Unused
  a: A, b: A --> (a >> b): vector<A>
  a: vector<A>, b: A --> (a >> b): vector<A>
  a: A, b: vector<A> --> (a >> b): vector<A>
  a: vector<A>, b: vector<A> --> (a >> b): vector<A>
``` |
| **Expect (a > b)** | ```
  a: A, b: B --> (a > b): tuple<A, B>
  a: A, b: Unused --> (a > b): A
  a: Unused, b: B --> (a > b): B
  a: Unused, b: Unused --> (a > b): Unused
  a: A, b: A --> (a > b): vector<A>
  a: vector<A>, b: A --> (a > b): vector<A>
  a: A, b: vector<A> --> (a > b): vector<A>
  a: vector<A>, b: vector<A> --> (a > b): vector<A>
``` |
| **Alternative (a | b)** | ```
  a: A, b: B --> (a | b): variant<A, B>
  a: A, b: Unused --> (a | b): optional<A>
  a: A, b: B, c: Unused --> (a | b | c): optional<variant<A, B>>
  a: Unused, b: B --> (a | b): optional<B>
  a: Unused, b: Unused --> (a | b): Unused
  a: A, b: A --> (a | b): A
``` |
| **Difference (a - b)** | ```
  a: A, b: B --> (a - b): A
  a: Unused, b: B --> (a - b): Unused
``` |
| **Kleene (+a)** | ```
  a: A --> +a: vector<A>
  a: Unused --> +a: Unused
``` |
| **Plus (+a)** | ```
  a: A --> +a: vector<A>
  a: Unused --> +a: Unused
``` |
| **List (a % b)** | ```
  a: A, b: B --> (a % b): vector<A>
  a: Unused, b: B --> (a % b): Unused
``` |
| **repeat(...,...)[a]** | ```
  a: A --> repeat(...,...)[a]: vector<A>
  a: Unused --> repeat(...,...)[a]: Unused
``` |
### Expression

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
</table>
| **Sequential Or** (a || b) | a: A, b: B --> (a || b): tuple<optional<A>, optional<B>>  
  a: A, b: Unused --> (a || b): optional<A>  
  a: Unused, b: B --> (a || b): optional<B>  
  a: Unused, Unused --> (a || b): Unused  
  a: A, b: A --> (a || b): vector<optional<A>> |
| **Optional** (-a)  | a: A --> -a: optional<A>  
  a: Unused --> -a: Unused |
| &a                 | a: A --> &a: Unused |
| !b                 | a: A --> !a: Unused |
| **Permutation** (a ^ b) | a: A, b: B --> (a ^ b): tuple<optional<A>, optional<B>>  
  a: A, b: Unused --> (a ^ b): optional<A>  
  a: Unused, b: B --> (a ^ b): optional<B>  
  a: Unused, Unused --> (a ^ b): Unused  
  a: A, b: A --> (a ^ b): vector<optional<A>> |

### Nonterminals

#### Notation

- **RT**
  Synthesized attribute. The rule or grammar's return type.
- **Arg1, Arg2, ArgN**
  Inherited attributes. Zero or more arguments.
- **L1, L2, LN**
  Zero or more local variables.
- **r, r2**
  Rules
- **g**
  A grammar
- **p**
  A parser expression
- **my_grammar**
  A user defined grammar

#### Terminology

- **Signature**
  RT(Arg1, Arg2 ... ,ArgN). The signature specifies the synthesized (return value) and inherited (arguments) attributes.
- **Locals**
  locals<L1, L2 ... , LN>. The local variables.
- **Skipper**
  The skip-parser type

#### Template Arguments

- **Iterator**
  The iterator type you will use for parsing.
A1, A2, A3 can be one of 1) Signature 2) Locals 3) Skipper.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule&lt;Iterator, A1, A2, A3&gt; r(name);</td>
<td>Rule declaration. <strong>Iterator</strong> is required. A1, A2, A3 are optional and can be specified in any order. <strong>name</strong> is an optional string that gives the rule its name, useful for debugging and error handling.</td>
</tr>
<tr>
<td>rule&lt;Iterator, A1, A2, A3&gt; r(r2);</td>
<td>Copy construct rule <strong>r</strong> from rule <strong>r2</strong>.</td>
</tr>
<tr>
<td>r = r2;</td>
<td>Assign rule <strong>r2</strong> to <strong>r</strong>.</td>
</tr>
<tr>
<td>r.alias()</td>
<td>return an alias of <strong>r</strong>. The alias is a parser that holds a reference to <strong>r</strong>. Reference semantics.</td>
</tr>
<tr>
<td>r.copy()</td>
<td>Get a copy of <strong>r</strong>.</td>
</tr>
<tr>
<td>r.name(name)</td>
<td>Naming a rule</td>
</tr>
<tr>
<td>r.name()</td>
<td>Getting the name of a rule</td>
</tr>
<tr>
<td>debug(r)</td>
<td>Debug rule <strong>r</strong></td>
</tr>
<tr>
<td>r = p;</td>
<td>Rule definition</td>
</tr>
<tr>
<td>r %= p;</td>
<td>Auto-rule definition. The attribute of <strong>p</strong> should be compatible with the synthesized attribute of <strong>r</strong>. When <strong>p</strong> is successful, its attribute is automatically propagated to <strong>r</strong>’s synthesized attribute.</td>
</tr>
<tr>
<td>template&lt;typename Iterator&gt; struct my_grammar : grammar&lt;Iterator, A1, A2, A3&gt; {</td>
<td>Grammar definition. <strong>name</strong> is an optional string that gives the grammar its name, useful for debugging and error handling.</td>
</tr>
<tr>
<td>my_grammar() : my_grammar::base_type(start, name)</td>
<td></td>
</tr>
<tr>
<td>// Rule definitions</td>
<td></td>
</tr>
<tr>
<td>start = /* ... */;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>rule&lt;Iterator, A1, A2, A3&gt; start;</td>
<td></td>
</tr>
<tr>
<td>// more rule declarations...</td>
<td></td>
</tr>
<tr>
<td>g.name(name)</td>
<td>Naming a grammar</td>
</tr>
<tr>
<td>g.name()</td>
<td>Getting the name of a grammar</td>
</tr>
</tbody>
</table>

**Parser Semantic Actions**

Has the form:

```
p[f]
```

where **f** is a function with the signatures:
void f(Attrib const&);
void f(Attrib const&, Context&);
void f(Attrib const&, Context&, bool&);

You can use Boost.Bind to bind member functions. For function objects, the allowed signatures are:

void operator()(Attrib const&, unused_type, unused_type) const;
void operator()(Attrib const&, Context&, unused_type) const;
void operator()(Attrib const&, Context&, bool&) const;

The unused_type is used in the signatures above to signify 'don't care'.

For more detailed information about semantic actions see: here.

**Phoenix**

Boost.Phoenix makes it easier to attach semantic actions. You just inline your lambda expressions:

```
p[phoenix-lambda-expression]
```

Spirit.Qi provides some Boost.Phoenix placeholders to important information from the Attrib and Context that are otherwise fiddly to extract.

**Spirit.Qi specific Phoenix placeholders**

1. **_1, _2... , _N**
   - Nth attribute of p
2. **_val**
   - The enclosing rule's synthesized attribute.
3. **_r1, _r2... , _rN**
   - The enclosing rule's Nth inherited attribute.
4. **_a, _b... , _j**
   - The enclosing rule's local variables (_a refers to the first).
5. **_pass**
   - Assign false to _pass to force a parser failure.

**Important**

All placeholders mentioned above are defined in the namespace boost::spirit and, for your convenience, are available in the namespace boost::spirit::qi as well.

For more detailed information about semantic actions see: here.

**Reference**

**Parser Concepts**

Spirit.Qi parsers fall into a couple of generalized concepts. The **Parser** is the most fundamental concept. All Spirit.Qi parsers are models of the **Parser** concept. **PrimitiveParser**, **UnaryParser**, **BinaryParser**, **NaryParser**, and **Nonterminal** are all refinements of the **Parser** concept.

The following sections provide details on these concepts.
Parser

Description

The Parser is the most fundamental concept. A Parser has a member function, parse, that accepts a first-last ForwardIterator pair and returns bool as its result. The iterators delimit the data being parsed. The Parser's parse member function returns true if the parse succeeds, in which case the first iterator is advanced accordingly. Each Parser can represent a specific pattern or algorithm, or it can be a more complex parser formed as a composition of other Parsers.

Notation

p A Parser.
P A Parser type.
Iter a ForwardIterator type.
f, l ForwardIterator. first/last iterator pair.
Context The parser's Context type.
context The parser's Context, or unused.
skip A skip Parser, or unused.
attrib A Compatible Attribute, or unused.

Valid Expressions

In the expressions below, the behavior of the parser, p, and how skip and attrib are handled by p, are left unspecified in the base Parser concept. These are specified in subsequent, more refined concepts and by the actual models thereof.

For any Parser the following expressions must be valid:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p).parse(f, l, context, skip, attrib)</td>
<td>Match the input sequence starting from ( f). Return true if successful, otherwise return false.</td>
<td>bool</td>
</tr>
<tr>
<td>p.what(context)</td>
<td>Get information about a Parser.</td>
<td>info</td>
</tr>
</tbody>
</table>

Type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P::template attribute&lt;Context, Iter&gt;::type</td>
<td>The Parser's expected attribute.</td>
</tr>
<tr>
<td>traits::is_parser&lt;P&gt;::type</td>
<td>Metaclass that evaluates to mpl::true_ if a certain type, P is a Parser, mpl::false_ otherwise (See MPL Boolean Constant).</td>
</tr>
</tbody>
</table>

Postcondition

Upon return from p.parse the following post conditions should hold:

- On a successful match, \( f\) is positioned one past the last matching character/token.
On a failed match, if a skip parser is unused, f is restored to its original position prior to entry.

On a failed match, if a skip parser is not unused, f is positioned one past the last character/token matching skip.

On a failed match, attrib state is undefined.

No post-skips: trailing skip characters/tokens will not be skipped.

Models

All parsers in Spirit.Qi are models of the Parser concept.

**PrimitiveParser**

**Description**

PrimitiveParser is the most basic building block that the client uses to build more complex parsers.

**Refinement of**

Parser

**Pre-skip**

Upon entry to the parse member function, a PrimitiveParser is required to do a pre-skip. Leading skip characters/tokens will be skipped prior to parsing. Only PrimitiveParsers are required to perform this pre-skip. This is typically carried out through a call to qi::skip_over:

```cpp
qi::skip_over(f, l, skip);
```

**Type Expressions**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>traits::is_primitive_parser&lt;P&gt;::type</td>
<td>Metafunction that evaluates to mpl::true_ if a certain type, P, is a PrimitiveParser, mpl::false_ otherwise (See MPL Boolean Constant).</td>
</tr>
</tbody>
</table>

**Models**

- attr(attrib)
- eoi
- eol
- eps
- symbols<>

**UnaryParser**

**Description**

UnaryParser is a composite parser that has a single subject. The UnaryParser may change the behavior of its subject following the Delegate Design Pattern.

**Refinement of**

Parser
Notation

\( p \)  A UnaryParser.
\( P \)  A UnaryParser type.

Valid Expressions

In addition to the requirements defined in Parser, for any UnaryParser the following must be met:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p.\text{subject} )</td>
<td>Subject parser.</td>
<td>Parser</td>
</tr>
</tbody>
</table>

Type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P::\text{subject_type} )</td>
<td>The subject parser type.</td>
</tr>
<tr>
<td>( \text{traits::is_unary_parser}&lt;P&gt;::\text{type} )</td>
<td>Metafunction that evaluates to \text{mpl::true_} if a certain type, ( P ) is a UnaryParser, \text{mpl::false_} otherwise (See MPL Boolean Constant).</td>
</tr>
</tbody>
</table>

Invariants

For any UnaryParser, \( p \), the following invariant always holds:

- \( \text{traits::is\_parser}<P::\text{subject\_type}>::\text{type} \text{ evaluates to } \text{mpl::true\_} \)

Models

- And Predicate
- Kleene
- \text{lexeme}
- Not Predicate
- omit
- Plus
- \text{raw}
- repeat
- skip

BinaryParser

Description

\text{BinaryParser} is a composite parser that has a two subjects, \text{left} and \text{right}. The BinaryParser allows its subjects to be treated in the same way as a single instance of a \text{Parser} following the Composite Design Pattern.

Refinement of

Parser
Notation

\[ p \quad \text{A BinaryParser.} \]

\[ P \quad \text{A BinaryParser type.} \]

Valid Expressions

In addition to the requirements defined in \textit{Parser}, for any BinaryParser the following must be met:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p.left )</td>
<td>Left parser.</td>
<td>\textit{Parser}</td>
</tr>
<tr>
<td>( p.right )</td>
<td>Right parser.</td>
<td>\textit{Parser}</td>
</tr>
</tbody>
</table>

Type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P::\text{left_type} )</td>
<td>The left parser type.</td>
</tr>
<tr>
<td>( P::\text{right_type} )</td>
<td>The right parser type.</td>
</tr>
<tr>
<td>( \text{traits::is_binary_parser&lt;P&gt;::type} )</td>
<td>Metafunction that evaluates to \textit{mpl::true} if a certain type, ( P ) is a BinaryParser, \textit{mpl::false} otherwise (See MPL Boolean Constant).</td>
</tr>
</tbody>
</table>

Invariants

For any BinaryParser, \( P \), the following invariants always hold:

- \( \text{traits::is\_parser<P::\text{left\_type}>::type} \) evaluates to \textit{mpl::true}.
- \( \text{traits::is\_parser<P::\text{right\_type}>::type} \) evaluates to \textit{mpl::true}.

Models

- Difference
- List

\textbf{NaryParser}

\textbf{Description}

\textit{NaryParser} is a composite parser that has one or more subjects. The NaryParser allows its subjects to be treated in the same way as a single instance of a \textit{Parser} following the Composite Design Pattern.

\textbf{Refinement of}

\textit{Parser}

\textbf{Notation}

\[ p \quad \text{A NaryParser.} \]

\[ P \quad \text{A NaryParser type.} \]
Valid Expressions

In addition to the requirements defined in `Parser`, for any NaryParser the following must be met:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
<th>Return type</th>
</tr>
</thead>
</table>

Type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p.elements_type</td>
<td>Elements tuple type.</td>
</tr>
<tr>
<td>traits::is_nary_parser&lt;P&gt;::type</td>
<td>Metafunction that evaluates to mpl::true_ if a certain type,</td>
</tr>
<tr>
<td></td>
<td>P is a NaryParser, mpl::false_ otherwise (See MPL Boolean</td>
</tr>
<tr>
<td></td>
<td>Constant).</td>
</tr>
</tbody>
</table>

Invariants

For each element, `E`, in any NaryParser, `P`, the following invariant always holds:

- `traits::is_parser<E>::type` evaluates to `mpl::true_`

Models

- Alternative
- Expect
- Permutation
- Sequence
- Sequential Or

Nonterminal

Description

A Nonterminal is a symbol in a Parsing Expression Grammar production that represents a grammar fragment. Nonterminals may self reference to specify recursion. This is one of the most important concepts and the reason behind the word "recursive" in recursive descent parsing.

Refinement of `Parser`

Signature

Nonterminals can have both synthesized and inherited attributes. The Nonterminal's `Signature` specifies both the synthesized and inherited attributes. The specification uses the function declarator syntax:

```
RT(A0, A1, A2, ..., AN)
```

where `RT` is the Nonterminal's synthesized attribute and `A0` ... `AN` are the Nonterminal's inherited attributes.
**Attributes**

The Nonterminal models a C++ function. The Nonterminal's synthesized attribute is analogous to the function return value and its inherited attributes are analogous to function arguments. The inherited attributes (arguments) can be passed in just like any Lazy Argument, e.g.:

```cpp
r(expr) // Evaluate expr at parse time and pass the result to the Nonterminal r
```

`_val`

The `boost::spirit::qi::_val` placeholder can be used in Phoenix semantic actions anywhere in the Nonterminal's definition. This Phoenix placeholder refers to the Nonterminal's (synthesized) attribute. The `_val` placeholder acts like a mutable reference to the Nonterminal's attribute.

**Note**

Starting with Spirit V2.5 (distributed with Boost V1.47) the placeholder `_val` can be used in semantic actions attached to top level parser components as well. See The Parse API for more information.

`_r1 ... _r10`

The `boost::spirit::_r1 ... boost::spirit::r10` placeholders can be used in Phoenix semantic actions anywhere in the Nonterminal's definition. These Phoenix placeholders refer to the Nonterminal's inherited attributes.

**Locals**

Nonterminals can have local variables that will be created on the stack at parse time. A locals descriptor added to the Nonterminal declaration will give the Nonterminal local variables:

```cpp
template <typename T0, typename T1, typename T2, ..., typename TN>
struct locals;
```

where `T0 ... TN` are the types of local variables accessible in your Phoenix semantic actions using the placeholders:

- `boost::spirit::_a`
- `boost::spirit::_b`
- `boost::spirit::_c`
- `boost::spirit::_d`
- `boost::spirit::_e`
- `boost::spirit::_f`
- `boost::spirit::_g`
- `boost::spirit::_h`
- `boost::spirit::_i`
- `boost::spirit::_j`

which correspond to the Nonterminal's local variables `T0 ... T9`. 
Notation

\( x \)  
A Nonterminal

\( x \)  
A Nonterminal type

\( \text{arg1, \text{arg2}, ..., \text{argN}} \)  
Lazy Arguments that evaluate to each of the Nonterminal's inherited attributes.

Valid Expressions

In addition to the requirements defined in Parser, for any Nonterminal the following must be met:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>In a parser expression, invoke Nonterminal ( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>( x(\text{arg1, arg2, ..., argN}) )</td>
<td>In a parser expression, invoke Nonterminal ( x ) passing in inherited attributes ( \text{arg1} ) ... ( \text{argN} )</td>
<td>( x )</td>
</tr>
<tr>
<td>( x.\text{name}(\text{name}) )</td>
<td>Naming a Nonterminal.</td>
<td>void</td>
</tr>
<tr>
<td>( x.\text{name}() )</td>
<td>Getting the name of a Nonterminal.</td>
<td>std::string</td>
</tr>
<tr>
<td>\text{debug}(x) )</td>
<td>Debug Nonterminal ( x ).</td>
<td>void</td>
</tr>
</tbody>
</table>

Type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X::\text{sig_type} )</td>
<td>The Signature of ( X ): An MPL Forward Sequence. The first element is the Nonterminal's synthesized attribute type and the rest are the inherited attribute types.</td>
</tr>
<tr>
<td>( X::\text{locals_type} )</td>
<td>The local variables of ( X ): An MPL Forward Sequence.</td>
</tr>
</tbody>
</table>

Models

- rule
- grammar

Parser Basics

Lazy Argument

Some parsers (e.g. primitives and non-terminals) may take in additional attributes. Such parsers take the form:

\[
p(a_1, a_2, ..., a_N)
\]

where \( p \) is a parser. Each of the arguments \( (a_1 \) ... \( a_N \)) can either be an immediate value, or a function, \( f \), with signature:

\[
T f(\text{Unused}, \text{Context})
\]

where \( T \), the function's return value, is compatible with the argument type expected and \( \text{Context} \) is the parser's Context type (The first argument is \textit{unused} to make the \( \text{Context} \) the second argument. This is done for uniformity with Semantic Actions).
Character Encoding Namespace

Some parsers need to know which character set a char or wchar_t is operating on. For example, the \texttt{alnum} parser works differently with ISO8859.1 and ASCII encodings. Where necessary, Spirit encodes (tags) the parser with the character set.

We have a namespace for each character set Spirit will be supporting. That includes \texttt{ascii}, \texttt{iso8859_1}, \texttt{standard} and \texttt{standard_wide} (and in the future, \texttt{unicode}). In each of the character encoding namespaces, we place tagged versions of parsers such as \texttt{alnum}, \texttt{space} etc.

Example:

\begin{verbatim}
using boost::spirit::ascii::space; // use the ASCII space parser
\end{verbatim}

Namespaces:

\begin{itemize}
\item boost::spirit::ascii
\item boost::spirit::iso8859_1
\item boost::spirit::standard
\item boost::spirit::standard_wide
\end{itemize}

For ease of use, the components in this namespaces are also brought into the qi sub-namespaces with the same names:

\begin{itemize}
\item boost::spirit::qi::ascii
\item boost::spirit::qi::iso8859_1
\item boost::spirit::qi::standard
\item boost::spirit::qi::standard_wide
\end{itemize}

Examples

All sections in the reference present some real world examples. The examples use a common test harness to keep the example code as minimal and direct to the point as possible. The test harness is presented below.

Some includes:

\begin{verbatim}
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/adapt_struct.hpp>
#include <boost/assert.hpp>
#include <iostream>
#include <string>
#include <cstdlib>
\end{verbatim}

Our test functions:

These functions test the parsers without attributes.
template <typename P>
void test_parser(
    char const* input, P const& p, bool full_match = true)
{
    using boost::spirit::qi::parse;

    char const* f(input);
    char const* l(f + strlen(f));
    if (parse(f, l, p) && (!full_match || (f == l)))
        std::cout << "ok" << std::endl;
    else
        std::cout << "fail" << std::endl;
}

template <typename P>
void test_phrase_parser(
    char const* input, P const& p, bool full_match = true)
{
    using boost::spirit::qi::phrase_parse;
    using boost::spirit::qi::ascii::space;

    char const* f(input);
    char const* l(f + strlen(f));
    if (phrase_parse(f, l, p, space) && (!full_match || (f == l)))
        std::cout << "ok" << std::endl;
    else
        std::cout << "fail" << std::endl;
}

These functions test the parsers with user supplied attributes.

template <typename P, typename T>
void test_parser_attr(
    char const* input, P const& p, T& attr, bool full_match = true)
{
    using boost::spirit::qi::parse;

    char const* f(input);
    char const* l(f + strlen(f));
    if (parse(f, l, p, attr) && (!full_match || (f == l)))
        std::cout << "ok" << std::endl;
    else
        std::cout << "fail" << std::endl;
}

template <typename P, typename T>
void test_phrase_parser_attr(
    char const* input, P const& p, T& attr, bool full_match = true)
{
    using boost::spirit::qi::phrase_parse;
    using boost::spirit::qi::ascii::space;

    char const* f(input);
    char const* l(f + strlen(f));
    if (phrase_parse(f, l, p, space, attr) && (!full_match || (f == l)))
        std::cout << "ok" << std::endl;
    else
        std::cout << "fail" << std::endl;
}

The print_info utility function prints information contained in the info class.
struct printer
{
    typedef boost::spirit::utf8_string string;

    void element(string const& tag, string const& value, int depth) const
    {
        for (int i = 0; i < (depth*4); ++i) // indent to depth
            std::cout << ' ';

        std::cout << "tag: " << tag;
        if (value !="")
            std::cout << ", value: " << value;
        std::cout << std::endl;
    }
};

void print_info(boost::spirit::info const& what)
{
    using boost::spirit::basic_info_walker;

    printer pr;
    basic_info_walker<printer> walker(pr, what.tag, 0);
    boost::apply_visitor(walker, what.value);
}

String

A string can be any object \texttt{s}, of type \texttt{S}, that satisfies the following expression traits:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{boost::spirit::traits::is_string\langle S\rangle}</td>
<td>Metafunction that evaluates to \texttt{mpl::true_} if a certain type, \texttt{S} is a string, \texttt{mpl::false_} otherwise (See MPL Boolean Constant).</td>
</tr>
<tr>
<td>\texttt{boost::spirit::traits::char_type_of\langle S\rangle}</td>
<td>Metafunction that returns the underlying char type of a string type, \texttt{S}.</td>
</tr>
<tr>
<td>\texttt{boost::spirit::traits::get_c_string(s)}</td>
<td>Function that returns the underlying raw C-string from \texttt{s}.</td>
</tr>
<tr>
<td>\texttt{boost::spirit::traits::get_begin(s)}</td>
<td>Function that returns an STL iterator from \texttt{s} that points to the beginning the string.</td>
</tr>
<tr>
<td>\texttt{boost::spirit::traits::get_end(s)}</td>
<td>Function that returns an STL iterator from \texttt{s} that points to the end of the string.</td>
</tr>
</tbody>
</table>

Models

Predefined models include:

- any literal string, e.g. "Hello, World",
- a pointer/reference to a null-terminated array of characters
a `std::basic_string<Char>`

The namespace `boost::spirit::traits` is open for users to provide their own specializations. The customization points implemented by `Spirit.Qi` usable to customize the behavior of parsers are described in the section [Customization of Attribute Handling](#).

## Parser API

### Iterator Based Parser API

### Description

The library provides a couple of free functions to make parsing a snap. These parser functions have two forms. The first form `parse` works on the character level. The second `phrase_parse` works on the phrase level and requires skip parser. Both versions can take in attributes by reference that will hold the parsed values on a successful parse.

### Header

```
#include <boost/spirit/include/qi_parse.hpp>
```

For variadic attributes:

```
#include <boost/spirit/include/qi_parse_attr.hpp>
```

The variadic attributes version of the API allows one or more attributes to be passed into the parse functions. The functions taking two or more are usable when the parser expression is a `Sequence` only. In this case each of the attributes passed have to match the corresponding part of the sequence.

For the API functions deducing the correct (matching) parser type from the supplied attribute type:

```
#include <boost/spirit/include/qi_parse_auto.hpp>
```

Also, see [Include Structure](#).

### Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boost::spirit::qi::parse</code></td>
</tr>
<tr>
<td><code>boost::spirit::qi::phrase_parse</code></td>
</tr>
<tr>
<td><code>boost::spirit::qi::skip_flag::postskip</code></td>
</tr>
<tr>
<td><code>boost::spirit::qi::skip_flag::dont_postskip</code></td>
</tr>
</tbody>
</table>
namespace boost { namespace spirit { namespace qi {

template <typename Iterator, typename Expr>
inline bool
parse(
    Iterator& first
 , Iterator last
 , Expr const& expr);

template <typename Iterator, typename Expr,
    typename Attr1, typename Attr2, ..., typename AttrN>
inline bool
parse(
    Iterator& first
 , Iterator last
 , Expr const& expr
 , Attr1& attr1, Attr2& attr2, ..., AttrN& attrN);

template <typename Iterator, typename Expr, typename Skipper>
inline bool
phrase_parse(
    Iterator& first
 , Iterator last
 , Expr const& expr
 , Skipper const& skipper
 , BOOST_SCOPED_ENUM(skip_flag) post_skip = skip_flag::postskip);

template <typename Iterator, typename Expr, typename Skipper,
    typename Attr1, typename Attr2, ..., typename AttrN>
inline bool
phrase_parse(
    Iterator& first
 , Iterator last
 , Expr const& expr
 , Skipper const& skipper
 , Attr1& attr1, Attr2& attr2, ..., AttrN& attrN);

template <typename Iterator, typename Expr, typename Skipper,
    typename Attr1, typename Attr2, ..., typename AttrN>
inline bool
phrase_parse(
    Iterator& first
 , Iterator last
 , Expr const& expr
 , Skipper const& skipper
 , BOOST_SCOPED_ENUM(skip_flag) post_skip
 , Attr1& attr1, Attr2& attr2, ..., AttrN& attrN);
}}}

Note

Starting with Spirit V2.5 (distributed with Boost V1.47) the placeholder _val can be used in semantic actions attached to top level parser components. In this case _val refers to the supplied attribute as a whole. For API functions taking more than one attribute argument _val will refer to a Fusion vector or references to the attributes.

Spirit.Qi parser API functions based on the automatic creation of the matching parser type:
namespace boost { namespace spirit { namespace qi {

    template <typename Iterator, typename Attr>
    inline bool
    parse(  
        Iterator& first
        , Iterator last
        , Attr& attr);

    template <typename Iterator, typename Attr, typename Skipper>
    inline bool
    phrase_parse(  
        Iterator& first
        , Iterator last
        , Attr& attr
        , Skipper const& skipper
        , BOOST_SCOPED_ENUM(skip_flag) post_skip = skip_flag::postskip);

}}}

All functions above return true if none of the involved parser components failed, and false otherwise.

The maximum number of supported arguments is limited by the preprocessor constant SPIRIT_ARGUMENTS_LIMIT. This constant defaults to the value defined by the preprocessor constant PHOENIX_LIMIT (which in turn defaults to 10).

Note

The variadic functions with two or more attributes internally combine references to all passed attributes into a fusion::vector and forward this as a combined attribute to the corresponding one attribute function.

The phrase_parse functions not taking an explicit skip_flag as one of their arguments invoke the passed skipper after a successful match of the parser expression. This can be inhibited by using the other versions of that function while passing skip_flag::dont_postskip to the corresponding argument.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator</td>
<td>ForwardIterator pointing to the source to parse.</td>
</tr>
<tr>
<td>Expr</td>
<td>An expression that can be converted to a Qi parser.</td>
</tr>
<tr>
<td>Skipper</td>
<td>Parser used to skip white spaces.</td>
</tr>
<tr>
<td>Attr</td>
<td>An attribute type utilized to create the corresponding parser type from.</td>
</tr>
<tr>
<td>Attr1, Attr2, ..., AttrN</td>
<td>One or more attributes.</td>
</tr>
</tbody>
</table>

Stream Based Parser API

Description

The library provides a couple of Standard IO Manipulators allowing to integrate Spirit.Qi input parsing facilities with Standard input streams. These parser manipulators have two forms. The first form, match, works on the character level. The second phrase_match works on the phrase level and requires a skip parser. Both versions can take in attributes by reference that will hold the parsed values on a successful parse.
Header

// forwards to <boost/spirit/home/qi/stream/match_manip.hpp>
#include <boost/spirit/include/qi_match.hpp>

For variadic attributes:

// forwards to <boost/spirit/home/qi/stream/match_manip_attr.hpp>
#include <boost/spirit/include/qi_match_attr.hpp>

The variadic attributes version of the API allows one or more attributes to be passed into the parse manipulators. The manipulators taking two or more attributes are usable when the parser expression is a Sequence only. In this case each of the attributes passed have to match the corresponding part of the sequence.

For the API functions deducing the correct (matching) parser type from the supplied attribute type:

// forwards to <boost/spirit/home/qi/match_auto.hpp>
#include <boost/spirit/include/qi_match_auto.hpp>

Also, see Include Structure.

Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::qi::match</td>
</tr>
<tr>
<td>boost::spirit::qi::phrase_match</td>
</tr>
<tr>
<td>boost::spirit::qi::skip_flag::postskip</td>
</tr>
<tr>
<td>boost::spirit::qi::skip_flag::dont_postskip</td>
</tr>
</tbody>
</table>
Synopsis

namespace boost { namespace spirit { namespace qi {

    template <typename Expr>
    inline <unspecified>
    match(
        Expr const& xpr);

    template <typename Expr, typename Attr1, typename Attr2, ..., typename AttrN>
    inline <unspecified>
    match(
        Expr const& xpr,
        Attr1& attr1, Attr2& attr2, ..., AttrN& attrN);

    template <typename Expr, typename Skipper>
    inline <unspecified>
    phrase_match(
        Expr const& expr,
        Skipper const& s,
        BOOST_SCOPED_ENUM(skip_flag) post_skip = skip_flag::postskip);

    template <typename Expr, typename Skipper, typename Attr1, typename Attr2, ..., typename AttrN>
    inline <unspecified>
    phrase_match(
        Expr const& expr,
        Skipper const& s,
        Attr1& attr1, Attr2& attr2, ..., AttrN& attrN);

}}

Spirit.Qi parser API functions based on the automatic creation of the matching parser type:

namespace boost { namespace spirit { namespace qi {

    template <typename Attr>
    inline <unspecified>
    match(
        Attr& attr);

    template <typename Attr, typename Skipper>
    inline <unspecified>
    phrase_match(
        Attr& attr,
        Skipper const& s,
        BOOST_SCOPED_ENUM(skip_flag) post_skip = skip_flag::postskip);

}}

All functions above return a standard IO stream manipulator instance (see Manipulators), which when streamed from an input stream will result in parsing the input using the embedded Spirit.Qi parser expression. Any error (or failed parse) occurring during the invocation of the Spirit.Qi parsers will be reflected in the streams status flag (std::ios_base::failbit will be set).
The maximum number of supported arguments is limited by the preprocessor constant `SPIRIT_ARGUMENTS_LIMIT`. This constant defaults to the value defined by the preprocessor constant `PHOENIX_LIMIT` (which in turn defaults to 10).

**Note**

The variadic manipulators with two or more attributes internally combine references to all passed attributes into a `fusion::vector` and forward this as a combined attribute to the corresponding manipulator taking one attribute.

The `phrase_match` manipulators not taking an explicit `skip_flag` as one of their arguments invoke the passed skipper after a successful match of the parser expression. This can be inhibited by using the other versions of that manipulator while passing `skip_flag::dont_postskip` to the corresponding argument.

**Template parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>An expression that can be converted to a Qi parser.</td>
</tr>
<tr>
<td>Skipper</td>
<td>Parser used to skip white spaces.</td>
</tr>
<tr>
<td>Attr</td>
<td>An attribute type utilized to create the corresponding parser type from.</td>
</tr>
<tr>
<td>Attr1, Attr2, ..., AttrN</td>
<td>One or more attributes.</td>
</tr>
</tbody>
</table>

**API for Automatic Parser Creation**

**Description**

The library implements a special API returning a parser instance for a supplied attribute type. This function finds the best matching parser type for the attribute based on a set of simple matching rules (as outlined in the table below) applied recursively to the attribute type. The returned parser can be utilized to match input for the provided attribute.

**Header**

```cpp
// forwards to <boost/spirit/home/qi/auto.hpp>
#include <boost/spirit/include/qi_auto.hpp>
```

Also, see Include Structure.

**Namespace**

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::qi::create_parser</td>
</tr>
<tr>
<td>boost::spirit::traits::create_parser_exists</td>
</tr>
</tbody>
</table>
Synopsis

```cpp
namespace boost { namespace spirit { namespace qi
{
  template <typename Attr>
  inline <unspecified>
  create_parser();
}}}
```

The returned instance can be directly passed as the parser (or the skipping parser) to any of the Spirit.Qi API functions. Additionally it can be assigned to a rule as the rules right hand side expression. This function will return a valid parser type only if the meta function `traits::create_parser_exists` returns `mpl::true_`. Otherwise it will fail compiling.

```cpp
namespace boost { namespace spirit { namespace traits
{
  template <typename Attr>
  struct create_parser_exists;
}}}
```

The meta function evaluates to `mpl::true_` if `create_parser` would return a valid parser for the given type `Attr`.

The following table outlines the mapping rules from the attribute type to the parser type. These rules are applied recursively to create the parser type which can be used to match input for the given attribute type.

<table>
<thead>
<tr>
<th>Attribute type</th>
<th>Generator type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, wchar_t</td>
<td>standard::char_, standard::wide::char_</td>
</tr>
<tr>
<td>short, int, long</td>
<td>short_, int_, long_</td>
</tr>
<tr>
<td>unsigned short, unsigned int, unsigned long</td>
<td>ushort_, uint_, ulong_</td>
</tr>
<tr>
<td>float, double, long double</td>
<td>float_, double_, longdouble</td>
</tr>
<tr>
<td>short, int, long</td>
<td>short_, int_, long_</td>
</tr>
<tr>
<td>long long, unsigned long long</td>
<td>longlong, ullonglong</td>
</tr>
<tr>
<td>bool</td>
<td>bool_</td>
</tr>
<tr>
<td>Any (STL) container</td>
<td>Kleene Star (unary '*' )</td>
</tr>
<tr>
<td>Any Fusion sequence</td>
<td>Sequence operator ('&lt;&lt;')</td>
</tr>
<tr>
<td>boost::optional&lt;&gt;</td>
<td>Optional operator (unary ' - ' )</td>
</tr>
<tr>
<td>boost::variant&lt;&gt;</td>
<td>Alternative operator ('</td>
</tr>
</tbody>
</table>

**Important**

The mapping for the parsers `long long` and `ulong long` are only available on platforms where the preprocessor constant `BOOST_HAS_LONG_LONG` is defined (i.e. on platforms having native support for `long long` and `unsigned long long` (64 bit) signed and unsigned integer types).
Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attr</td>
<td>An attribute type utilized to create the corresponding parser type from.</td>
</tr>
</tbody>
</table>

Semantic Actions with Parsers

Description

Semantic actions may be attached to any point in the grammar specification. They allow to call a function or function object in order to provide the value to be output by the parser attached to the semantic action. Semantic actions are associated with a parser using the syntax `p[fa]`, where `p` is an arbitrary parser expression.

Header

```cpp
// forwards to <boost/spirit/home/qi/action.hpp>
#include <boost/spirit/include/qi_action.hpp>
```

Also, see Include Structure.

Model of

UnaryParser

Notation

- `a, p` Instances of a parser, `P`
- `A` Attribute type exposed by a parser, `a`
- `fa` A (semantic action) function with signature `void(Attrib&, Context, bool&)`. The third parameter is a boolean flag that can be set to false to force the parser to fail. Both `Context` and the boolean flag are optional. For more information see below.
- `Attrib` The attribute obtained from the parse.
- `Context` The type of the parser execution context. For more information see below.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p[fa]</code></td>
<td>If <code>p</code> is successful, call semantic action, <code>fa</code>. The function or function object <code>fa</code> is provided the attribute value parsed by the parser <code>p</code>, plus some more context information and a mutable bool flag which can be used to fail parsing.</td>
</tr>
</tbody>
</table>

The possible signatures for functions to be used as semantic actions are:
The function or function object is expected to return the value to generate output from by assigning it to the first parameter, `attr`. Here `Attrib` is the attribute type of the parser attached to the semantic action.

The type `Context` is the type of the parser execution context. This type is unspecified and depends on the context the parser is invoked in. The value, `context` used by semantic actions written using Phoenix to access various context dependent attributes and values. For more information about Phoenix placeholder expressions usable in semantic actions see Nonterminal.

The third parameter, `pass`, can be used by the semantic action to force the associated parser to fail. If `pass` is set to `false` the action parser will immediately return `false` as well, while not invoking `p` and not generating any output.

**Attributes**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a[fa]</code></td>
<td><code>a: A --&gt; a[fa]: A</code></td>
</tr>
</tbody>
</table>

**Complexity**

The complexity of the action parser is defined by the complexity of the parser the semantic action is attached to and the complexity of the function or function object used as the semantic action.

**Example**

Examples for semantic actions can be found here: Examples of Semantic Actions.

**Auto Parser**

**Description**

This module includes the description of the `auto_` parser. This parser can be used to automatically create a parser based on the supplied attribute type.

**Header**

```cpp
// forwards to <boost/spirit/home/qi/auto.hpp>
#include <boost/spirit/include/qi_auto.hpp>
```

Also, see Include Structure.

**Namespace**

**Name**

`boost::spirit::auto_ // alias: boost::spirit::qi::auto_`

**Model of**

`PrimitiveParser`
Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in `PrimitiveGenerator`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>auto_</code></td>
<td>Create a parser instance compatible with the supplied attribute type and use it for input matching.</td>
</tr>
</tbody>
</table>

Additional Requirements

The `auto_` parsers can be used to match input for any data type for which a mapping to a parser type is defined (the meta function `traits::create_parser_exists` returns `mpl::true_`). The following table outlines the predefined mapping rules from the attribute type to the parser type. These rules are applied recursively to create the parser type which can be used to match input for the given attribute type.

<table>
<thead>
<tr>
<th>Attribute type</th>
<th>Parser type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char, wchar_t</code></td>
<td><code>standard::char_, standard_wide::char_</code></td>
</tr>
<tr>
<td><code>short, int, long</code></td>
<td><code>short_, int_, long_</code></td>
</tr>
<tr>
<td><code>unsigned short, unsigned int, unsigned long</code></td>
<td><code>ushort_, uint_, ulong_</code></td>
</tr>
<tr>
<td><code>float, double, long double</code></td>
<td><code>float_, double_, long_double</code></td>
</tr>
<tr>
<td><code>long long, unsigned long long</code></td>
<td><code>long_long, ulong_long</code></td>
</tr>
<tr>
<td><code>bool</code></td>
<td><code>bool_</code></td>
</tr>
<tr>
<td>Any (STL) container</td>
<td>Kleene Star (unary <code>*</code>)</td>
</tr>
<tr>
<td>Any Fusion sequence</td>
<td>Sequence operator (<code>&gt;&gt;</code>)</td>
</tr>
<tr>
<td><code>boost::optional&lt;&gt;</code></td>
<td>Optional operator (unary '-')</td>
</tr>
<tr>
<td><code>boost::variant&lt;&gt;</code></td>
<td>Alternative operator (`</td>
</tr>
</tbody>
</table>

It is possible to add support for any custom data type by implementing a specialization of the customization point `traits::create_parser`. This customization can be used also to redefined any of the predefined mappings.

Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>auto_</code></td>
<td><code>hold_any</code></td>
</tr>
</tbody>
</table>

Important

The attribute type `hold_any` exposed by the `auto_` parser is semantically and syntactically equivalent to the type implemented by `Boost.Any`. It has been added to `Spirit` as it has better a performance and a smaller footprint if compared to `Boost.Any`. 
Complexity

The complexity of the auto parser depends on the supplied attribute type. Each attribute type results in a different parser type to be instantiated which defines the overall complexity.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/adapt_struct.hpp>
#include <boost/assert.hpp>
#include <iostream>
#include <string>
#include <cstdlib>
```

Some using declarations:

```
using boost::spirit::qi::auto_;  
```

And a class definition used in the examples:

```
// a simple complex number representation z = a + bi
struct complex
{
    complex (double a = 0.0, double b = 0.0)
        : a(a), b(b)
    {}

    double a;
    double b;
};
```

The following construct is required to allow the complex data structure to be utilized as a Boost.Fusion sequence. This is required as we will emit output for this data structure with a Spirit.Qi sequence: '{' >> qi::double_ >> ',' >> qi::double_ >> '}'.

```
BOOST_FUSION_ADAPT_STRUCT(
    complex,
    (double, a)
    (double, b)
)
```

We add a specialization for the create_parser customization point defining a custom output format for the complex type. Generally, any specialization for create_parser is expected to return the proto expression to be used to match input for the type the customization point has been specialized for.

We need to utilize proto::deep_copy as the expression contains literals (the '{', ',', '}', and '}') which normally get embedded in the proto expression by reference only. The deep copy converts the proto tree to hold this by value. The deep copy operation can
be left out for simpler proto expressions (not containing references to temporaries). Alternatively you could use the proto::make_expr facility to build the required proto expression.

```cpp
namespace boost { namespace spirit { namespace traits {
    template <>
    struct create_parser<complex>
    {
        typedef proto::result_of::deep_copy<
            BOOST_TYPEOF('{' >> qi::double_ >> ',' >> qi::double_ >> '}')
        >::type type;

        static type call()
        {
            return proto::deep_copy(
                '{' >> qi::double_ >> ',' >> qi::double_ >> '}');
        }
    };
}}}
```

Some usage examples of auto_ parsers:

Parse a simple integer using the generated parser component int_:

```cpp
int i = 0;
test_parser_attr("123", auto_, i);
std::cout << i << std::endl; // prints: 123
```

Parse an instance of the complex data type as defined above using the parser as generated by the defined customization point:

```cpp
complex c;
test_parser_attr("(1.2,2.4)", auto_, c);
std::cout << c.a << "," << c.b << std::endl; // prints: 1.2,2.4
```

### Auxiliary Parsers

This module includes different auxiliary parsers not fitting into any of the other categories. This module includes the attr, attr_cast, eoi, eol, eps, and lazy parsers.

#### Module Header

// forwards to <boost/spirit/home/qi/auxiliary.hpp>
#include <boost/spirit/include/qi_auxiliary.hpp>

Also, see Include Structure.

#### Attribute Parser (attr)

### Description

The Attribute parser does not consume any input, for this reason it always matches an empty string and always succeeds. It's purpose is to expose its specified parameter as an attribute.
Header

// forwards to <boost/spirit/home/qi/auxiliary/attr.hpp>
#include <boost/spirit/include/qi_attr.hpp>

Also, see Include Structure.

Namespace

Name

boost::spirit::attr // alias: boost::spirit::qi::attr

Model of

PrimitiveParser

Notation

a  A arbitrary typed constant value, e.g. 0.0, "Hello", or a variable of arbitrary type or a Lazy Argument that evaluates to an arbitrary type.

A  The type of a or if it is a Lazy Argument, its return type.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr(a)</td>
<td>Create a pseudo parser exposing the current value of a as its attribute without consuming any input at parse time.</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr(a)</td>
<td>A</td>
</tr>
</tbody>
</table>

Complexity

O(1)

The complexity is constant as no input is consumed and no matching is done.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:
namespace phx = boost::phoenix;
using boost::spirit::qi::attr;

Using attr with literals:

```cpp
test_parser_attr("", attr("boost"), str);
std::cout << str << std::endl;  // will print 'boost'

double d;
test_parser_attr("", attr(1.0), d);
std::cout << d << std::endl;  // will print '1.0'
```

Using attr with Phoenix function objects:

```cpp
d = 0.0;
double d1 = 1.2;
test_parser_attr("", attr(phx::ref(d1)), d);
std::cout << d << std::endl;  // will print '1.2'
```

### Attribute Transformation Pseudo Parser (attr_cast)

#### Description

The `attr_cast<Exposed, Transformed>()` component invokes the embedded parser while supplying an attribute of type `Transformed`. The supplied attribute gets created from the original attribute (of type `Exposed`) passed to this component using the customization point `traits::transform_attribute`.

#### Header

```cpp
// forwards to <boost/spirit/home/qi/auxiliary/attr_cast.hpp>
#include <boost/spirit/include/qi_attr_cast.hpp>
```

Also, see Include Structure.

#### Namespace

**Name**

`boost::spirit::attr_cast // alias: boost::spirit::qi::attr_cast`

#### Synopsis

```cpp
template <Exposed, Transformed>
<unspecified> attr_cast(<unspecified>);
```

#### Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exposed</td>
<td>The type of the attribute supplied to the attr_cast.</td>
<td>unused_type</td>
</tr>
<tr>
<td>Transformed</td>
<td>The type of the attribute expected by the embedded parser p.</td>
<td>unused_type</td>
</tr>
</tbody>
</table>
The `attr_cast` is a function template. It is possible to invoke it using the following schemes:

```cpp
attr_cast(p)
attr_cast<Exposed>(p)
attr_cast<Exposed, Transformed>(p)
```

depending on which of the attribute types can be deduced properly if not explicitly specified.

**Model of**

`UnaryParser`

**Notation**

- `p` A parser object.

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in `UnaryParser`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>attr_cast(p)</code></td>
<td>Create a component invoking the parser <code>p</code> while passing an attribute of the type as normally expected by <code>p</code>. The type of the supplied attribute will be transformed to the type <code>p</code> exposes as its attribute type (by using the attribute customization point <code>traits::transform_attribute</code>).</td>
</tr>
<tr>
<td><code>attr_cast&lt;Exposed&gt;(p)</code></td>
<td>Create a component invoking the parser <code>p</code> while passing an attribute of the type as normally expected by <code>p</code>. The supplied attribute is expected to be of the type <code>Exposed</code>, it will be transformed to the type <code>p</code> exposes as its attribute type (using the attribute customization point <code>traits::transform_attribute</code>).</td>
</tr>
<tr>
<td><code>attr_cast&lt;Exposed, Transformed&gt;(p)</code></td>
<td>Create a component invoking the parser <code>p</code> while passing an attribute of type <code>Transformed</code>. The supplied attribute is expected to be of the type <code>Exposed</code>, it will be transformed to the type <code>Transformed</code> (using the attribute customization point <code>traits::transform_attribute</code>).</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>attr_cast(p)</code></td>
<td><code>p: A --&gt; attr_cast(p): A</code></td>
</tr>
<tr>
<td><code>attr_cast&lt;Exposed&gt;(p)</code></td>
<td><code>p: A --&gt; attr_cast&lt;Exposed&gt;(p): Exposed</code></td>
</tr>
<tr>
<td><code>attr_cast&lt;Exposed, Transformed&gt;(p)</code></td>
<td><code>p: A --&gt; attr_cast&lt;Exposed, Transformed&gt;(p): Exposed</code></td>
</tr>
</tbody>
</table>

**Complexity**

The complexity of this component is fully defined by the complexity of the embedded parser `p`.
Some using declarations:

```cpp
using boost::spirit::qi::int_;`
```

The example references data structure `int_data` which needs a specialization of the customization point `traits::transform_attribute`:

```cpp
// this is just a test structure we want to use in place of an int
struct int_data
{
  int i;
};

// we provide a custom attribute transformation to allow its use as an int
namespace boost { namespace spirit { namespace traits
{
  // in this case we just expose the embedded 'int' as the attribute instance
  // to use, allowing to leave the function 'post()' empty
  template <>
  struct transform_attribute<int_data, int, qi::domain>
  {
    typedef int& type;
    static int& pre(int_data& d) { return d.i; }
    static void post(int_data& val, int const& attr) {}
    static void fail(int_data&) {}
  };
}}}
```

Now we use the `attr_cast` pseudo parser to invoke the attribute transformation:

```cpp
int_data d = { 0 };
test_parser_attr("1", boost::spirit::qi::attr_cast(int_), d);
std::cout << d.i << std::endl;
```

### End of Line Parser (`eol`)

#### Description

The `eol` parser matches the end of line (CR/LF and combinations thereof).

#### Header

```cpp
// forwards to <boost/spirit/home/qi/auxiliary/eol.hpp>
#include <boost/spirit/include/qi_eol.hpp>
```

Also, see Include Structure.
Namespace

Name

boost::spirit::eol // alias: boost::spirit::qi::eol

Model of

PrimitiveParser

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveParser.

Expression | Semantics
---|---
eol | Create a parser that matches the end of line.

Attributes

Expression | Attribute
---|---
eol | unused

Complexity

O(1)

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::qi::eol;
```

Using eol:

```cpp
test_parser("\n", eol);
```

End of Input Parser (eoi)

Description

The eoi parser matches the end of input (returns a successful match with 0 length when the input is exhausted)

Header

```cpp
// forwards to <boost/spirit/home/qi/auxiliary/eoi.hpp>
#include <boost/spirit/include/qi_eoi.hpp>
```

Also, see Include Structure.
Namespace

boost::spirit::eoi // alias: boost::spirit::qi::eoi

Model of

PrimitiveParser

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>eoi</td>
<td>Create a parser that matches the end of input.</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>eoi</td>
<td>unused</td>
</tr>
</tbody>
</table>

Complexity

O(1)

Example

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```
using boost::spirit::qi::eoi;
```

Using eoi:

```
test_parser("", eoi);
```

Epsilon Parser (eps)

**Description**

The Epsilon (eps) is a multi-purpose parser that returns a zero length match.

**Simple Form**

In its simplest form, eps matches the null string and always returns a match of zero length:
eps // always returns a zero-length match

This form is usually used to trigger a semantic action unconditionally. For example, it is useful in triggering error messages when a set of alternatives fail:

```
 r = a | b | c | eps[error()]; // Call error if a, b, and c fail to match
```

**Semantic Predicate**

Semantic predicates allow you to attach a conditional function anywhere in the grammar. In this role, the epsilon takes a Lazy Argument that returns `true` or `false`. The Lazy Argument is typically a test that is called to resolve ambiguity in the grammar. A parse failure will be reported when the Lazy Argument result evaluates to `false`. Otherwise an empty match will be reported. The general form is:

```
eps(f) >> rest;
```

The Lazy Argument `f` is called to do a semantic test (say, checking if a symbol is in the symbol table). If test returns `true`, `rest` will be evaluated. Otherwise, the production will return early with a no-match without ever touching `rest`.

**Header**

```
// forwards to <boost/spirit/home/qi/auxiliary/eps.hpp>
#include <boost/spirit/include/qi_eps.hpp>
```

Also, see Include Structure.

**Namespace**

**Name**

```
boost::spirit::eps // alias: boost::spirit::qi::eps
```

**Model of**

PrimitiveParser

**Notation**

`f` A Lazy Argument that evaluates `bool`.

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>eps</td>
<td>Match an empty string (always matches).</td>
</tr>
<tr>
<td>eps(f)</td>
<td>If <code>f</code> evaluates to <code>true</code>, return a zero length match.</td>
</tr>
</tbody>
</table>
Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>eps</td>
<td>unused</td>
</tr>
</tbody>
</table>

Complexity

For plain (eps) the complexity is O(1). For Semantic predicates (eps(f)) the complexity is defined by the function f.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::qi::eps;
using boost::spirit::qi::int_;
using boost::spirit::qi::_1;
namespace phx = boost::phoenix;
```

Basic eps:

```cpp
test_parser("", eps); // always matches
```

This example simulates the "classic" if_p parser. Here, int_ will be tried only if the condition, c, is true.

```cpp
bool c = true; // a flag
test_parser("1234", eps(phx::ref(c) == true) >> int_);
```

This example simulates the "classic" while_p parser. Here, the kleene loop will exit once the condition, c, becomes true. Notice that the condition, c, is turned to false when we get to parse 4.

```cpp
test_phrase_parser("1 2 3 4", 
   *(eps(phx::ref(c) == true) >> int_[phx::ref(c) = (_1 == 4)]);
```

Lazy Parser (lazy)

Description

The lazy parser, as its name suggests, invokes a lazy Phoenix function that returns a parser at parse time. This parser will be used once it is created to continue the parse.

Header

```cpp
// forwards to <boost/spirit/home/qi/auxiliary/lazy.hpp>
#include <boost/spirit/include/qi_lazy.hpp>
```

Also, see Include Structure.
Namespace

Name

boost::spirit::lazy // alias: boost::spirit::qi::lazy

Model of

Parser

Notation

fp  A Lazy Argument that evaluates to a Parser.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in Parser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>fp</td>
<td>Create a lazy-parser from a Lazy Argument, fp. fp will be invoked at parse time. fp is expected to return a Parser object. This parser is then invoked in order to parse the input.</td>
</tr>
<tr>
<td>lazy(fp)</td>
<td>Create a lazy-parser from a Lazy Argument, fp. fp will be invoked at parse time. fp is expected to return a Parser object. This parser is then invoked in order to parse the input.</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>fp</td>
<td>The attribute type of the return type of fp.</td>
</tr>
<tr>
<td>lazy(fp)</td>
<td>The attribute type of the return type of fp.</td>
</tr>
</tbody>
</table>

Complexity

The complexity of the lazy parser is determined by the complexity of the parser returned from fp.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::qi::lazy;
using boost::spirit::ascii::string;
using boost::phoenix::val;
```

Using lazy:
Here, the `phoenix::val` expression creates a function that returns its argument when invoked. The `lazy` expression defers the invocation of this function at parse time. Then, this parser (string parser) is called into action. All this takes place at parse time.

```c++
test_parser("Hello", lazy(val(string("Hello"))));
```

The above is equivalent to:

```c++
test_parser("Hello", val(string("Hello")));
```

## Binary Parsers

This module includes different parsers for parsing binary data in various endianness. It includes parsers for default (native), little, and big endian binary input.

### Module Header

```cpp
// forwards to <boost/spirit/home/qi/binary.hpp>
#include <boost/spirit/include/qi_binary.hpp>
```

Also, see [Include Structure](#).

### Binary Native Endianness Parsers

#### Description

Binary native endian parsers are designed to parse binary byte streams that are laid out in the native endianness of the target architecture.

#### Header

```cpp
// forwards to <boost/spirit/home/qi/binary.hpp>
#include <boost/spirit/include/qi_binary.hpp>
```

Also, see [Include Structure](#).

### Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boost::spirit::byte_</code> // alias: <code>boost::spirit::qi::byte_</code></td>
</tr>
<tr>
<td><code>boost::spirit::word</code> // alias: <code>boost::spirit::qi::word</code></td>
</tr>
<tr>
<td><code>boost::spirit::dword</code> // alias: <code>boost::spirit::qi::dword</code></td>
</tr>
<tr>
<td><code>boost::spirit::qword</code> // alias: <code>boost::spirit::qi::qword</code></td>
</tr>
</tbody>
</table>

#### Note

`qword` is only available on platforms where the preprocessor constant `BOOST_HAS_LONG_LONG` is defined (i.e. on platforms having native support for unsigned `long long` (64 bit) integer types).

### Model of

`PrimitiveParser`
### Notation

- **b**: A single byte (8 bit binary value) or a Lazy Argument that evaluates to a single byte. This value is always in native endian.

- **w**: A 16 bit binary value or a Lazy Argument that evaluates to a 16 bit binary value. This value is always in native endian.

- **dw**: A 32 bit binary value or a Lazy Argument that evaluates to a 32 bit binary value. This value is always in native endian.

- **qw**: A 64 bit binary value or a Lazy Argument that evaluates to a 64 bit binary value. This value is always in native endian.

### Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in `PrimitiveParser`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>byte_</code></td>
<td>Matches any 8 bit native endian binary.</td>
</tr>
<tr>
<td><code>word</code></td>
<td>Matches any 16 bit native endian binary.</td>
</tr>
<tr>
<td><code>dword</code></td>
<td>Matches any 32 bit native endian binary.</td>
</tr>
<tr>
<td><code>qword</code></td>
<td>Matches any 64 bit native endian binary.</td>
</tr>
<tr>
<td><code>byte_(b)</code></td>
<td>Matches an exact 8 bit native endian binary.</td>
</tr>
<tr>
<td><code>word(w)</code></td>
<td>Matches an exact 16 bit native endian binary.</td>
</tr>
<tr>
<td><code>dword(dw)</code></td>
<td>Matches an exact 32 bit native endian binary.</td>
</tr>
<tr>
<td><code>qword(qw)</code></td>
<td>Matches an exact 64 bit native endian binary.</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>byte_</code></td>
<td><code>boost::uint_least8_t</code></td>
</tr>
<tr>
<td><code>word</code></td>
<td><code>boost::uint_least16_t</code></td>
</tr>
<tr>
<td><code>dword</code></td>
<td><code>boost::uint_least32_t</code></td>
</tr>
<tr>
<td><code>qword</code></td>
<td><code>boost::uint_least64_t</code></td>
</tr>
<tr>
<td><code>byte_(b)</code></td>
<td><code>unused</code></td>
</tr>
<tr>
<td><code>word(w)</code></td>
<td><code>unused</code></td>
</tr>
<tr>
<td><code>dword(dw)</code></td>
<td><code>unused</code></td>
</tr>
<tr>
<td><code>qword(qw)</code></td>
<td><code>unused</code></td>
</tr>
</tbody>
</table>

### Complexity

O(N), where N is the number of bytes parsed
Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Using declarations and variables:

```cpp
using boost::spirit::qi::byte_; 
using boost::spirit::qi::word; 
using boost::spirit::qi::dword; 
using boost::spirit::qi::qword;

boost::uint8_t uc; 
boost::uint16_t us; 
boost::uint32_t ui; 
boost::uint64_t ul;
```

Basic usage of the native binary parsers for little endian platforms:

```cpp
test_parser_attr("\x01", byte_, uc); assert(uc == 0x01);
test_parser_attr("\x01\x02", word, us); assert(us == 0x0201);
test_parser_attr("\x01\x02\x03\x04", dword, ui); assert(ui == 0x04030201);
test_parser_attr("\x01\x02\x03\x04\x05\x06\x07\x08", qword, ul); 
assert(ul == 0x0807060504030201LL);

test_parser("\x01", byte_(0x01));
test_parser("\x01\x02", word(0x0102));
test_parser("\x01\x02\x03\x04", dword(0x04030201));
test_parser("\x01\x02\x03\x04\x05\x06\x07\x08", 
            qword(0x0807060504030201LL));
```

Basic usage of the native binary parsers for big endian platforms:

```cpp
test_parser_attr("\x01", byte_, uc); assert(uc == 0x01);
test_parser_attr("\x01\x02", word, us); assert(us == 0x0201);
test_parser_attr("\x01\x02\x03\x04", dword, ui); assert(ui == 0x01020304);
test_parser_attr("\x01\x02\x03\x04\x05\x06\x07\x08", qword, ul); 
assert(ul == 0x0102030405060708LL);

test_parser("\x01", byte_(0x01));
test_parser("\x01\x02", word(0x0102));
test_parser("\x01\x02\x03\x04", dword(0x01020304));
test_parser("\x01\x02\x03\x04\x05\x06\x07\x08", 
            qword(0x0102030405060708LL));
```

Binary Little Endianness Parser

Description

Binary little endian parsers are designed to parse binary byte streams that are laid out in little endian.
Header

// forwards to <boost/spirit/home/qi/binary.hpp>
#include <boost/spirit/include/qi_binary.hpp>

Also, see Include Structure.

Namespace

Name

boost::spirit::little_word // alias: boost::spirit::qi::little_word
boost::spirit::little_dword // alias: boost::spirit::qi::little_dword
boost::spirit::little_qword // alias: boost::spirit::qi::little_qword

Note

little_qword is only available on platforms where the preprocessor constant BOOST_HAS_LONG_LONG is defined (i.e. on platforms having native support for unsigned long long (64 bit) integer types).

Model of

PrimitiveParser

Notation

w A 16 bit binary value or a Lazy Argument that evaluates to a 16 bit binary value. This value is always in native endian.
dw A 32 bit binary value or a Lazy Argument that evaluates to a 32 bit binary value. This value is always in native endian.
qw A 64 bit binary value or a Lazy Argument that evaluates to a 64 bit binary value. This value is always in native endian.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>little_word</td>
<td>Matches any 16 bit little endian binary.</td>
</tr>
<tr>
<td>little_dword</td>
<td>Matches any 32 bit little endian binary.</td>
</tr>
<tr>
<td>little_qword</td>
<td>Matches any 64 bit little endian binary.</td>
</tr>
<tr>
<td>little_word(w)</td>
<td>Matches an exact 16 bit little endian binary.</td>
</tr>
<tr>
<td>little_dword(dw)</td>
<td>Matches an exact 32 bit little endian binary.</td>
</tr>
<tr>
<td>little_qword(qw)</td>
<td>Matches an exact 32 bit little endian binary.</td>
</tr>
</tbody>
</table>

Spirit 2.5

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/
### Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>little_word</td>
<td>boost::uint_least16_t</td>
</tr>
<tr>
<td>little_dword</td>
<td>boost::uint_least32_t</td>
</tr>
<tr>
<td>little_qword</td>
<td>boost::uint_least64_t</td>
</tr>
<tr>
<td>little_word(w)</td>
<td>unused</td>
</tr>
<tr>
<td>little_dword(dw)</td>
<td>unused</td>
</tr>
<tr>
<td>little_qword(qw)</td>
<td>unused</td>
</tr>
</tbody>
</table>

### Complexity

$O(N)$, where $N$ is the number of bytes parsed

### Example

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

Using declarations and variables:

```cpp
using boost::spirit::qi::little_word;
using boost::spirit::qi::little_dword;
using boost::spirit::qi::little_qword;

boost::uint16_t us;
boost::uint32_t ui;
boost::uint64_t ul;
```

Basic usage of the little endian binary parsers:

```cpp
test_parser_attr("\x01\x02", little_word, us); assert(us == 0x0201);
test_parser_attr("\x01\x02\x03\x04", little_dword, ui); assert(ui == 0x04030201);
test_parser_attr("\x01\x02\x03\x04\x05\x06\x07\x08", little_qword, ul);
assert(ul == 0x0807060504030201LL);
test_parser("\x01\x02", little_word(0x0201));
test_parser("\x01\x02\x03\x04", little_dword(0x04030201));
test_parser("\x01\x02\x03\x04\x05\x06\x07\x08", little_qword(0x0807060504030201LL));
```

### Binary Big Endianness Parsers

#### Description

Binary big endian parsers are designed to parse binary byte streams that are laid out in big endian.
Header

// forwards to <boost/spirit/home/qi(binary.hpp>
#include <boost/spirit/include/qi_binary.hpp>

Also, see Include Structure.

Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::big_word // alias: boost::spirit::qi::big_word</td>
</tr>
<tr>
<td>boost::spirit::big_dword // alias: boost::spirit::qi::big_dword</td>
</tr>
<tr>
<td>boost::spirit::big_qword // alias: boost::spirit::qi::big_qword</td>
</tr>
</tbody>
</table>

Note

big_qword is only available on platforms where the preprocessor constant BOOST_HAS_LONG_LONG is defined (i.e. on platforms having native support for unsigned long long (64 bit) integer types).

Model of

PrimitiveParser

Notation

<table>
<thead>
<tr>
<th>A 16 bit binary value or a Lazy Argument that evaluates to a 16 bit binary value. This value is always in native endian.</th>
</tr>
</thead>
<tbody>
<tr>
<td>big_word</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A 32 bit binary value or a Lazy Argument that evaluates to a 32 bit binary value. This value is always in native endian.</th>
</tr>
</thead>
<tbody>
<tr>
<td>big_dword</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A 64 bit binary value or a Lazy Argument that evaluates to a 64 bit binary value. This value is always in native endian.</th>
</tr>
</thead>
<tbody>
<tr>
<td>big_qword</td>
</tr>
</tbody>
</table>

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>big_word</td>
<td>Matches any 16 bit big endian binary.</td>
</tr>
<tr>
<td>big_dword</td>
<td>Matches any 32 bit big endian binary.</td>
</tr>
<tr>
<td>big_qword</td>
<td>Matches any 64 bit big endian binary.</td>
</tr>
<tr>
<td>big_word(w)</td>
<td>Matches an exact 16 bit big endian binary.</td>
</tr>
<tr>
<td>big_dword(dw)</td>
<td>Matches an exact 32 bit big endian binary.</td>
</tr>
<tr>
<td>big_qword(qw)</td>
<td>Matches an exact 32 bit big endian binary.</td>
</tr>
</tbody>
</table>
### Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>big_word</td>
<td>boost::uint_least16_t</td>
</tr>
<tr>
<td>big_dword</td>
<td>boost::uint_least32_t</td>
</tr>
<tr>
<td>big_qword</td>
<td>boost::uint_least64_t</td>
</tr>
<tr>
<td>big_word(w)</td>
<td>unused</td>
</tr>
<tr>
<td>big_dword(dw)</td>
<td>unused</td>
</tr>
<tr>
<td>big_qword(qw)</td>
<td>unused</td>
</tr>
</tbody>
</table>

### Complexity

$O(N)$, where $N$ is the number of bytes parsed

### Example

#### Note

The test harness for the example(s) below is presented in the Basics Examples section.

Using declarations and variables:

```cpp
using boost::spirit::qi::big_word;
using boost::spirit::qi::big_dword;
using boost::spirit::qi::big_qword;

boost::uint16_t us;
boost::uint32_t ui;
boost::uint64_t ul;
```

Basic usage of the big endian binary parsers:

```cpp
test_parser_attr("\x01\x02", big_word, us); assert(us == 0x0102);
test_parser_attr("\x01\x02\x03\x04", big_dword, ui); assert(ui == 0x01020304);
test_parser_attr("\x01\x02\x03\x04\x05\x06\x07\x08", big_qword, ul);
assert(0x0102030405060708LL);
```

```cpp
test_parser("\x01\x02", big_word(0x0102));
test_parser("\x01\x02\x03\x04", big_dword(0x01020304));
test_parser("\x01\x02\x03\x04\x05\x06\x07\x08",
            big_qword(0x0102030405060708LL));
```

### Character Parsers

This module includes parsers for single characters. Currently, this module includes literal chars (e.g. 'x', L'x'), char_ (single characters, ranges and character sets) and the encoding specific character classifiers (alnum, alpha, digit, xdigit, etc.).
Module Header

```cpp
// forwards to <boost/spirit/home/qi/char.hpp>
#include <boost/spirit/include/qi_char.hpp>
```

Also, see Include Structure.

**Character Parser (char_, lit)**

**Description**

The char_ parser matches single characters. The char_ parser has an associated Character Encoding Namespace. This is needed when doing basic operations such as inhibiting case sensitivity and dealing with character ranges.

There are various forms of char_.

**char_**

The no argument form of char_ matches any character in the associated Character Encoding Namespace.

```cpp
char_  // matches any character
```

**char_(ch)**

The single argument form of char_ (with a character argument) matches the supplied character.

```cpp
char_('x')  // matches 'x'
char_(L'x') // matches L'x'
char_(x)    // matches x (a char)
```

**char_**(first, last)

char_ with two arguments, matches a range of characters.

```cpp
char_('a','z')  // alphabetic characters
char_(L'0',L'9') // digits
```

A range of characters is created from a low-high character pair. Such a parser matches a single character that is in the range, including both endpoints. Note, the first character must be before the second, according to the underlying Character Encoding Namespace.

Character mapping is inherently platform dependent. It is not guaranteed in the standard for example that 'A' < 'Z', that is why in Spirit2, we purposely attach a specific Character Encoding Namespace (such as ASCII, ISO-8859-1) to the char_ parser to eliminate such ambiguities.

**Note**

**Sparse bit vectors**

To accommodate 16/32 and 64 bit characters, the char-set statically switches from a std::bitset implementation when the character type is not greater than 8 bits, to a sparse bit/boolean set which uses a sorted vector of disjoint ranges (range_run). The set is constructed from ranges such that adjacent or overlapping ranges are coalesced.

range_runs are very space-economical in situations where there are lots of ranges and a few individual disjoint values. Searching is O(log n) where n is the number of ranges.
char_(def)

Lastly, when given a string (a plain C string, a std::basic_string, etc.), the string is regarded as a char-set definition string following a syntax that resembles posix style regular expression character sets (except that double quotes delimit the set elements instead of square brackets and there is no special negation ^ character). Examples:

<table>
<thead>
<tr>
<th>char_(&quot;a-zA-Z&quot;)</th>
<th>// alphabetic characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>char_(&quot;0-9a-fA-F&quot;)</td>
<td>// hexadecimal characters</td>
</tr>
<tr>
<td>char_(&quot;actgACTG&quot;)</td>
<td>// DNA identifiers</td>
</tr>
<tr>
<td>char_(&quot;\x7f\x7e&quot;)</td>
<td>// Hexadecimal 0x7F and 0x7E</td>
</tr>
</tbody>
</table>

lit(ch)

lit, when passed a single character, behaves like the single argument char_ except that lit does not synthesize an attribute. A plain char or wchar_t is equivalent to a lit.

Note

lit is reused by both the string parsers and the char parsers. In general, a char parser is created when you pass in a character and a string parser is created when you pass in a string. The exception is when you pass a single element literal string, e.g. lit("x"). In this case, we optimize this to create a char parser instead of a string parser.

Examples:

'x'
lit('x')
lit(L'x')
lit(c) // c is a char

Header

// forwards to <boost/spirit/home/qi/char/char.hpp>
#include <boost/spirit/include/qi_char_.hpp>

Also, see Include Structure.

Namespace

Name

boost::spirit::lit // alias: boost::spirit::qi::lit
ns::char_

In the table above, ns represents a Character Encoding Namespace.

Model of

PrimitiveParser

Notation

c, f, l  A literal char, e.g. 'x', L'x' or anything that can be converted to a char or wchar_t, or a Lazy Argument that evaluates to anything that can be converted to a char or wchar_t.

ns  A Character Encoding Namespace.
A String or a Lazy Argument that evaluates to a String that specifies a char-set definition string following a syntax that resembles posix style regular expression character sets (except the square brackets and the negation ^ character).

A char parser, a char range parser or a char set parser.

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Create char parser from a char, c.</td>
</tr>
<tr>
<td>lit(c)</td>
<td>Create a char parser from a char, c.</td>
</tr>
<tr>
<td>ns::char_</td>
<td>Create a char parser that matches any character in the ns encoding.</td>
</tr>
<tr>
<td>ns::char_(c)</td>
<td>Create a char parser with ns encoding from a char, c.</td>
</tr>
<tr>
<td>ns::char_(f, l)</td>
<td>Create a char-range parser that matches characters from range (f to l, inclusive) with ns encoding.</td>
</tr>
<tr>
<td>ns::char_(cs)</td>
<td>Create a char-set parser with ns encoding from a char-set definition string, cs.</td>
</tr>
<tr>
<td>~cp</td>
<td>Negate cp. The result is a negated char parser that matches any character in the ns encoding except the characters matched by cp.</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>unused or if c is a Lazy Argument, the character type returned by invoking it.</td>
</tr>
<tr>
<td>lit(c)</td>
<td>unused or if c is a Lazy Argument, the character type returned by invoking it.</td>
</tr>
<tr>
<td>ns::char_</td>
<td>The character type of the Character Encoding Namespace, ns.</td>
</tr>
<tr>
<td>ns::char_(c)</td>
<td>The character type of the Character Encoding Namespace, ns.</td>
</tr>
<tr>
<td>ns::char_(f, l)</td>
<td>The character type of the Character Encoding Namespace, ns.</td>
</tr>
<tr>
<td>ns::char_(cs)</td>
<td>The character type of the Character Encoding Namespace, ns.</td>
</tr>
<tr>
<td>~cp</td>
<td>The attribute of cp.</td>
</tr>
</tbody>
</table>

**Complexity**

O(N), except for char-sets with 16-bit (or more) characters (e.g. wchar_t). These have O(log N) complexity, where N is the number of distinct character ranges in the set.
**Example**

---

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::qi::lit;
using boost::spirit::ascii::char_;```

Basic literals:

```cpp
test_parser("x", 'x'); // plain literal
test_parser("x", lit('x')); // explicit literal
test_parser("x", char_('x')); // ascii::char_
```

Range:

```cpp
char ch;
test_parser_attr("5", char_('0','9'), ch); // ascii::char_ range
std::cout << ch << std::endl; // prints '5'
```

Character set:

```cpp
test_parser_attr("5", char_("0-9"), ch); // ascii::char_ set
std::cout << ch << std::endl; // prints '5'
```

Lazy char_ using Phoenix

```cpp
namespace phx = boost::phoenix;
test_parser("x", phx::val('x')); // direct
test_parser("5",
       char_(phx::val('0'),phx::val('9'))); // ascii::char_ range
```

**Character Classification Parsers (alnum, digit, etc.)**

**Description**

The library has the full repertoire of single character parsers for character classification. This includes the usual alnum, alpha, digit, xdigit, etc. parsers. These parsers have an associated Character Encoding Namespace. This is needed when doing basic operations such as inhibiting case sensitivity.

**Header**

```cpp
#include <boost/spirit/home/qi/char/char_class.hpp>
```

Also, see Include Structure.
Namespace

<table>
<thead>
<tr>
<th>Name</th>
<th>ns::alnum</th>
<th>ns::alpha</th>
<th>ns::blank</th>
<th>ns::cntrl</th>
<th>ns::digit</th>
<th>ns::graph</th>
<th>ns::lower</th>
<th>ns::print</th>
<th>ns::punct</th>
<th>ns::space</th>
<th>ns::upper</th>
<th>ns::xdigit</th>
</tr>
</thead>
</table>

In the table above, ns represents a Character Encoding Namespace.

Model of

PrimitiveParser

Notation

ns  A Character Encoding Namespace.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveParser.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns::alnum</td>
<td>Matches alpha-numeric characters</td>
</tr>
<tr>
<td>ns::alpha</td>
<td>Matches alphabetic characters</td>
</tr>
<tr>
<td>ns::blank</td>
<td>Matches spaces or tabs</td>
</tr>
<tr>
<td>ns::cntrl</td>
<td>Matches control characters</td>
</tr>
<tr>
<td>ns::digit</td>
<td>Matches numeric digits</td>
</tr>
<tr>
<td>ns::graph</td>
<td>Matches non-space printing characters</td>
</tr>
<tr>
<td>ns::lower</td>
<td>Matches lower case letters</td>
</tr>
<tr>
<td>ns::print</td>
<td>Matches printable characters</td>
</tr>
<tr>
<td>ns::punct</td>
<td>Matches punctuation symbols</td>
</tr>
<tr>
<td>ns::space</td>
<td>Matches spaces, tabs, returns, and newlines</td>
</tr>
<tr>
<td>ns::upper</td>
<td>Matches upper case letters</td>
</tr>
<tr>
<td>ns::xdigit</td>
<td>Matches hexadecimal digits</td>
</tr>
</tbody>
</table>

**Attributes**

The character type of the Character Encoding Namespace, ns.

**Complexity**

O(N)

**Example**

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::ascii::alnum;
using boost::spirit::ascii::blank;
using boost::spirit::ascii::digit;
using boost::spirit::ascii::lower;
```

Basic usage:
Parser Directives

This module includes different directives usable to augment and parameterize other parsers. It includes the no_case, lexeme, omit, raw, repeat, matches, no_skip, skip, hold, as<T>, as_string and as_wstring directives.

Module Header

```
// forwards to <boost/spirit/home/qi/directive.hpp>
#include <boost/spirit/include/qi_directive.hpp>
```

Also, see Include Structure.

Parser Directive Inhibiting Skipping (lexeme[])

Description

The lexeme directive makes its subject a primitive. In a logical point of view, lexemes (and primitives) are minimal atomic units (e.g. words, numbers, identifiers, etc). These are the things that you'd normally put in the lexer (hinting at the term "lexeme"), but in a lexer-less world, you put these in a lexeme. Seeing its subject as a primitive, the lexeme directive does an initial pre-skip (as all primitives do) and turns off white space skipping.

At the phrase level, the parser ignores white spaces, possibly including comments. Use lexeme in situations where you want to work at the character level instead of the phrase level. Parsers can be made to work at the character level by enclosing the pertinent parts inside the lexeme directive. For example, here's a rule that parses integers:

```
integer = lexeme[ -(lit('+') | '-' ) >> +digit ];
```

The lexeme directive instructs its subject parser to work on the character level. Without it, the integer rule would have allowed erroneous embedded white spaces in inputs such as "1 2 345" which will be parsed as "12345".

Note

Keep in mind that lexeme[] pre-skips spaces. If this is not desired, use the no_skip directive instead.

Header

```
// forwards to <boost/spirit/home/qi/directive/lexeme.hpp>
#include <boost/spirit/include/qi_lexeme.hpp>
```

Also, see Include Structure.

Namespace

Name

```
boost::spirit::lexeme // alias: boost::spirit::qi::lexeme
```

Model of

UnaryParser
Notation

A Parser.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>lexeme[a]</td>
<td>Pre-skip and turn off white space skipping for the subject parser, a (and all its children).</td>
</tr>
</tbody>
</table>

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
</table>
| lexeme[a]  | a: A --> lexeme[a]: A  
a: Unused --> lexeme[a]: Unused |

Complexity

The complexity is defined by the complexity of the subject parser, a

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::qi::lexeme;
using boost::spirit::qi::lit;
using boost::spirit::ascii::digit;
```

Simple usage of lexeme[]:

The use of lexeme here will prevent skipping in between the digits and the sign making inputs such as "1 2 345" erroneous.

```cpp
test_phrase_parser("12345", lexeme( -(lit( '+' ) | '-' ) >> +digit ));
```

Parser Directive Inhibiting Skipping Without Pre-skip (no_skip[])

Description

The no_skip[] directive turns off white space skipping. The difference to lexeme is that it does not do pre-skipping in any case. Otherwise it is completely equivalent to the lexeme directive.
Header

```cpp
#include <boost/spirit/include/qi_no_skip.hpp>
```

Also, see Include Structure.

Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::no_skip // alias: boost::spirit::qi::no_skip</td>
</tr>
</tbody>
</table>

Model of

UnaryParser

Notation

a A Parser.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>no_skip[a]</td>
<td>Turns off white space skipping for the subject parser, a (and all its children). This directive does not pre-skips.</td>
</tr>
</tbody>
</table>

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>no_skip[a]</td>
<td>a: A --&gt; no_skip[a]: A</td>
</tr>
<tr>
<td></td>
<td>a: Unused --&gt; no_skip[a]: Unused</td>
</tr>
</tbody>
</table>

Complexity

The complexity is defined by the complexity of the subject parser, a

Example

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:
using boost::spirit::qi::no_skip;
using boost::spirit::qi::char_;  

Simple usage of `no_skip[]`:

The use of `no_skip` here will prevent skipping of whitespace in front and in between the characters of the string `' abc '`.  

```cpp
std::string str;
test_phrase_parser_attr("' abc '",
   '\'' >> no_skip[+~char_('\')] >> '\'', str);
std::cout << str << std::endl; // will output: > abc <
```

**Parser Directive Inhibiting Case Sensitivity (`no_case[]`)**

**Description**

The `no_case[]` directive does not consume any input. The actual matching is done by its subject parser. It's purpose is to force matching of the subject parser (and all its children) to be case insensitive.  

**Header**

```cpp
// forwards to <boost/spirit/home/qi/directive/no_case.hpp>
#include <boost/spirit/include/qi_no_case.hpp>
```

Also, see Include Structure.  

**Namespace**

**Name**

```cpp
ns::no_case
```

In the table above, `ns` represents a Character Encoding Namespace.  

**Model of**

The model of `no_case` is the model of its subject parser.  

**Notation**

a  A Parser.  

ns  A Character Encoding Namespace.  

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in the subject's concept.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ns::no_case[a]</code></td>
<td>Force matching of the subject parser, a (and all its children) to be case insensitive</td>
</tr>
</tbody>
</table>

**Attributes**

See Compound Attribute Notation.
Expression | Attribute
---|---
ns::no_case[a] | a: A --> ns::no_case[a]: A
a: Unused --> ns::no_case[a]: Unused

**Complexity**

The complexity is defined by the complexity of the subject parser, `a`

**Example**

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::ascii::no_case;
using boost::spirit::ascii::char_;  
using boost::spirit::ascii::alnum;
using boost::spirit::qi::symbols;
```

Simple usage of `no_case[]`:

```cpp
test_parser("X", no_case[char_('x')]);
test_parser("6", no_case[alnum]);
```

A more sophisticated use case of `no_case[]` in conjunction with a symbol table (see `symbols<>` for more details):

```cpp
symbols<char, int> sym;

sym.add
  ("apple", 1) // symbol strings are added in lowercase...
  ("banana", 2)
  ("orange", 3)
;  

int i;
// ...because sym is used for case-insensitive parsing

test_parser_attr("Apple", no_case[ sym ], i);
std::cout << i << std::endl;

test_parser_attr("ORANGE", no_case[ sym ], i);
std::cout << i << std::endl;
```

**Parser Directive Ignoring Attribute (omit[])**

**Description**

The `omit[]` ignores the attribute of its subject parser replacing it with `unused`. 

---

**Spirit 2.5**

**XML to PDF by RenderX XEP XSL-FO Formatter, visit us at [http://www.renderx.com/](http://www.renderx.com/)**
Header

// forwards to <boost/spirit/home/qi/directive/omit.hpp>
#include <boost/spirit/include/qi_omit.hpp>

Also, see Include Structure.

Namespace

Name

boost::spirit::omit // alias: boost::spirit::qi::omit

Model of

UnaryParser

Notation

a A Parser.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>omit[a]</td>
<td>Ignore the attribute of the subject parser, a</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>omit[a]</td>
<td>unused_type</td>
</tr>
</tbody>
</table>

Complexity

The complexity is defined by the complexity of the subject parser, a

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::qi::omit;
using boost::spirit::qi::int_;
using boost::spirit::ascii::char_;```

This parser ignores the first two characters and extracts the succeeding int:
```cpp
int i;
test_parser_attr("xx345", omit[char_ >> char_] >> int_, i);
std::cout << i << std::endl; // should print 345
```

**Directive for Transduction Parsing (raw[])**

**Description**

The `raw[]` disregards the attribute of its subject parser, instead exposing the half-open range `[first, last)` pointing to the matched characters from the input stream. The `raw[]` directive brings back the classic Spirit transduction (un-attributed) behavior for a subject parser.

**Header**

```cpp
// forwards to <boost/spirit/home/qi/directive/raw.hpp>
#include <boost/spirit/include/qi_raw.hpp>
```

Also, see Include Structure.

**Namespace**

**Name**

```cpp
boost::spirit::raw // alias: boost::spirit::qi::raw
```

**Model of**

UnaryParser

**Notation**

```cpp
a A Parser.
Iter A ForwardIterator type.
```

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in UnaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>raw[a]</td>
<td>Disregard the attribute of the subject parser, a. Expose instead the half-open range <code>[first, last)</code> pointing to the matched characters from the input stream.</td>
</tr>
</tbody>
</table>

**Attributes**

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
</table>
| raw[a]     | a: A --> raw[a]: boost::iterator_range<Iter>  
a: Unused --> raw[a]: Unused |
Note

See boost::iterator_range.

Complexity

The complexity is defined by the complexity of the subject parser, a

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::qi::raw;
using boost::spirit::ascii::alpha;
using boost::spirit::ascii::alnum;
```

This parser matches and extracts C++ identifiers:

```cpp
std::string id;
test_parser_attr("James007", raw[(alpha | '_') >> *(alnum | '_')], id);
std::cout << id << std::endl; // should print James007
```

Repetition Parser Directive (repeat[])

Description

The repeat[] provides a more powerful and flexible mechanism for repeating a parser. There are grammars that are impractical and cumbersome, if not impossible, for the basic EBNF iteration syntax (Kleene and the Plus) to specify. Examples:

- A file name may have a maximum of 255 characters only.
- A specific bitmap file format has exactly 4096 RGB color information.
- A 256 bit binary string (1..256 1s or 0s).

Header

```cpp
// forwards to <boost/spirit/home/qi/directive/repeat.hpp>
#include <boost/spirit/include/qi_repeat.hpp>
```

Also, see Include Structure.

Namespace

Name

```cpp
boost::spirit::repeat // alias: boost::spirit::qi::repeat
boost::spirit::inf // alias: boost::spirit::qi::inf
```
Model of
UnaryParser

Notation

\[ a \rightarrow A \text{ Parser.} \]

\[ n, \text{min, max} \rightarrow \text{An int anything that can be converted to an int, or a Lazy Argument that evaluates to anything that can be converted to an int.} \]

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeat[a]</td>
<td>Repeat a zero or more times. Same as Kleene.</td>
</tr>
<tr>
<td>repeat(n)[a]</td>
<td>Repeat a exactly ( n ) times.</td>
</tr>
<tr>
<td>repeat(min, max)[a]</td>
<td>Repeat a at least ( \text{min} ) times and at most ( \text{max} ) times.</td>
</tr>
<tr>
<td>repeat(min, inf)[a]</td>
<td>Repeat a at least ( \text{min} ) or more (continuing until a fails or the input is consumed).</td>
</tr>
</tbody>
</table>

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeat[a]</td>
<td>a: A \rightarrow \text{repeat}[a]: vector(&lt;A&gt;) \newline a: Unused \rightarrow \text{repeat}[a]: Unused</td>
</tr>
<tr>
<td>repeat(n)[a]</td>
<td>a: A \rightarrow \text{repeat}(n)[a]: vector(&lt;A&gt;) \newline a: Unused \rightarrow \text{repeat}(n)[a]: Unused</td>
</tr>
<tr>
<td>repeat(min, max)[a]</td>
<td>a: A \rightarrow \text{repeat}(\text{min, max})[a]: vector(&lt;A&gt;) \newline a: Unused \rightarrow \text{repeat}(\text{min, max})[a]: Unused</td>
</tr>
<tr>
<td>repeat(min, inf)[a]</td>
<td>a: A \rightarrow \text{repeat}(\text{min, inf})[a]: vector(&lt;A&gt;) \newline a: Unused \rightarrow \text{repeat}(\text{min, inf})[a]: Unused</td>
</tr>
</tbody>
</table>

Complexity

The overall complexity is defined by the complexity of its subject parser. The complexity of \text{repeat} itself is \( O(N) \), where \( N \) is the number of repetitions to execute.
Example

**Note**
The test harness for the example(s) below is presented in the Basics Examples section.

Using the repeat directive, we can now write our examples above.

Some using declarations:

```cpp
using boost::spirit::qi::repeat;
using boost::spirit::qi::lit;
using boost::spirit::qi::uint_parser;
using boost::spirit::qi::_1;
using boost::spirit::ascii::char_;
namespace phx = boost::phoenix;
```

A parser for a file name with a maximum of 255 characters:

```cpp
test_parser("batman.jpeg", repeat(1, 255)[char_("a-zA-Z_./")]);
```

A parser for a specific bitmap file format which has exactly 4096 RGB color information. (for the purpose of this example, we will be testing only 3 RGB color information.)

```cpp
uint_parser<unsigned, 16, 6, 6> rgb;
std::vector<unsigned> colors;
test_parser_attr("ffffff0000003f3f3f", repeat(3)[rgb], colors);
std::cout
   << std::hex
   << colors[0] << ',
   << colors[1] << ',
   << colors[2] << std::endl;
```

A 256 bit binary string (1..256 1s or 0s). (For the purpose of this example, we will be testing only 16 bits.)

```cpp
test_parser("1011010101110010", repeat(16)[lit('1') | '0']);
```

The Loop parsers can be dynamic. Consider the parsing of a binary file of Pascal-style length prefixed string, where the first byte determines the length of the incoming string. Here's a sample input:

```
11 H e l l o
W o r l d
```

This trivial example cannot be practically defined in traditional EBNF. Although some EBNF variants allow more powerful repetition constructs other than the Kleene Star, we are still limited to parsing fixed strings. The nature of EBNF forces the repetition factor to be a constant. On the other hand, Spirit allows the repetition factor to be variable at run time. We could write a grammar that accepts the input string above. Example using phoenix:
std::string str;
int n;
test_parser_attr("\x0bHello World",
    char_[phx::ref(n) = _1] >> repeat(phx::ref(n))|char_], str);
std::cout << n << ',' << str << std::endl; // will print "11,Hello World"

Directive Testing if Parser Succeeded \(\text{matches[]}\)

Description

The \text{matches[]} directive executes the embedded parser and returns whether it succeeded matching.

Header

```cpp
// forwards to <boost/spirit/home/qi/directive/matches.hpp>
#include <boost/spirit/include/qi_matches.hpp>
```

Also, see Include Structure.

Namespace

Name

```
boost::spirit::matches // alias: boost::spirit::qi::matches
```

Model of

\text{UnaryParser}

Notation

\(a\) A \text{Parser}.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in \text{UnaryParser}.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{matches}{}</td>
<td>Execute the subject parser (a), and return as its attribute whether it succeeded. The directive itself does always succeed.</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{matches}{}</td>
<td>bool</td>
</tr>
</tbody>
</table>

Complexity

The complexity is defined by the complexity of the subject parser, \(a\)
Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::qi::matches;
using boost::spirit::qi::int_;  
```

This parser tries to match an `int` and returns `true` as its attribute as it succeeded matching:

```cpp
bool result = false;
test_parser_attr("345", matches[int_], result);
std::cout << std::boolalpha << result << std::endl; // should print: true
```

This parser tries to match an `int` as well and returns `false` as its attribute as it fails matching:

```cpp
result = true;
test_parser_attr("abc", matches[int_], result);
std::cout << std::boolalpha << result << std::endl; // should print: false
```

**Parser Directive Re-Establishing Skipping (`skip[]`)**

**Description**

The `skip` directive is the inverse of `lexeme` or `qi_no_skip no_skip[]`. While the `lexeme` directive turns off white space skipping, the `skip` directive turns it on again. This is simply done by wrapping the parts inside the `skip` directive:

```cpp
skip[a]  
```

It is also possible to supply a skip parser to the `skip` directive:

```cpp
skip(p)[a] // Use `p` as a skipper for parsing `a`  
```

This makes it possible to:

- Perform localized phrase level parsing while doing character level parsing.
- Replace the current skipper anywhere with an entirely different skipper while doing phrase level parsing.

**Header**

```cpp
// forwards to <boost/spirit/home/qi/directive/skip.hpp>
#include <boost/spirit/include/qi_skip.hpp>
```

Also, see Include Structure.

**Namespace**

**Name**

```cpp
boost::spirit::skip // alias: boost::spirit::qi::skip
```
Model of

*UnaryParser*

**Notation**

a A Parser.

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in *UnaryParser*.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip[a]</td>
<td>Re-establish the skipper that got inhibited by lexeme or no_skip</td>
</tr>
<tr>
<td>skip(p)[a]</td>
<td>Use p as a skipper for parsing a</td>
</tr>
</tbody>
</table>

**Attributes**

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip[a]</td>
<td>a: A --&gt; skip[a]: A</td>
</tr>
<tr>
<td></td>
<td>a: Unused --&gt; skip[a]: Unused</td>
</tr>
<tr>
<td>skip(p)[a]</td>
<td>a: A --&gt; skip(p)[a]: A</td>
</tr>
<tr>
<td></td>
<td>a: Unused --&gt; skip(p)[a]: Unused</td>
</tr>
</tbody>
</table>

**Complexity**

The complexity is defined by the complexity of the subject parser, a

**Example**

*Note*

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::qi::skip;
using boost::spirit::qi::int_;
using boost::spirit::ascii::space;
```

Simple usage of `skip[]`:

Explicitly specify a skip parser. This parser parses comma delimited numbers, ignoring spaces.
Parser Directive for Attribute Commit/Rollback (hold[[]])

Description

The hold[[]] directive helps managing attributes, mainly for alternative parsers. It instantiates a new attribute instance for the embedded parser. The value of that attribute instance is copied to the outer attribute if the embedded parser succeeds and it is discarded otherwise. Alternative parsers normally do not rollback changes made to the outer attribute by an failed alternative. Wrapping those alternatives into a hold[[]] directive ensures that only the succeeding alternative gets to modify the attribute.

Header

```
// forwards to <boost/spirit/home/qi/directive/hold.hpp>
#include <boost/spirit/include/qi_hold.hpp>
```

Also, see Include Structure.

Namespace

Name

```
boost::spirit::hold // alias: boost::spirit::qi::hold
```

Model of

```
UnaryParser
```

Notation

```
a A Parser.
```

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>hold[a]</td>
<td>Create a new attribute instance while parsing a, copying the result to the outer attribute only after a succeeds.</td>
</tr>
</tbody>
</table>

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
</table>
| hold[a]    | a: A --> hold[a]: A  
a: Unused --> hold[a]: Unused |
Note

The hold[] directive uses swap() to implement the rollback/commit semantics for the attribute. For this reason the attribute type needs to to be usable with boost::swap (needs to either define a proper overload for swap(attribute_type&, attribute_type&) or expose a member function attribute_type::swap(attribute_type&).

Complexity

The complexity is defined by the complexity of the subject parser, a

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::qi::hold;
using boost::spirit::qi::int_;  
using boost::spirit::qi::attr;
```

The use of hold[] here will make sure the changes to the attribute caused by the (failing) first alternative will not be visible after the whole parsing succeeded.

```cpp
std::vector<int> v;
test_phrase_parser_attr("123",
    hold[int_ >> ':' >> int_] | int_ >> attr(0), v);
std::cout << v[0] << "," << v[1] << std::endl;  // will output: >123,0<
```

Parser Directives Forcing Atomic Assignment (as<T>, as_string[], as_wstring[])

Description

The as<T> class forces the atomic assignment of it's subject's synthesized attribute. Usually, repetitive parsers (such as Kleene, etc) or sequences exposing a vector<A> will assign elements to the container supplied as their synthesized attribute by calling traits::push_back_container repeatedly. In some cases, this may be undesirable. The as<T> class creates a directive that will pass a temporary object of type T to it's subject. If the subject parser passes, the temporary object will be assigned to the directive's supplied attribute with a single call to traits::assign_to. If the subject parser fails, the directive's attribute is not mutated.

Note

T is required to be a container type. If traits::is_container does not return true for T, a compile-time error will occur.

Note

The as<T> implicitly causes commit/rollback semantics similar in nature to the hold directive.
Caution

The `traits::assign_to` customization point may end up using `traits::push_back_container` to assign the temporary object to the supplied attribute by default, depending on the types involved. Use the interface described in Customization of Attribute Handling to manipulate the semantics of this assignment operation.

Header

```cpp
// forwards to <boost/spirit/home/qi/directive/as.hpp>
#include <boost/spirit/include/qi_as.hpp>
```

Also, see Include Structure.

Namespace

```cpp
namespace boost { namespace spirit {
    namespace qi {
        namespace as {
```

Synopsis

```cpp
template <typename T>
struct as;
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>A container type.</td>
<td>[none]</td>
</tr>
</tbody>
</table>

Model of

UnaryParser

Notation

- `a` A `Parser`.
- `t` A container of type `T`.
- `attr` The attribute supplied to the directive.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryParser.
Expression | Semantics
---|---
`as<T>()[a]` | Create a temporary object of `t` of type `T`, and invoke the subject parser `a`, supplying `t` as an attribute. If the subject parser passes, assign `t` to `attr`.

`as_string[a]` | Equivalent to `as<std::string>()[a]`

`as_wstring[a]` | Equivalent to `as<std::wstring>()[a]`

**Attributes**

See [Compound Attribute Notation](#).

**Expression** | **Attribute**
---|---
`as<T>()[a]` | `a: A --> as<T>()[a]: T`

**Complexity**

The complexity is defined by the complexity of the subject parser, `a`, and the complexity of the assignment of the container `t` to the supplied attribute `attr`.

**Example**

**Note**

The test harness for the example(s) below is presented in the [Basics Examples](#) section.

Some using declarations:

```cpp
using boost::spirit::utree;
using boost::spirit::utree_type;
using boost::spirit::utf8_symbol_type;
using boost::spirit::qi::as;
using boost::spirit::qi::as_string;
using boost::spirit::qi::char_;```

Simple usage of `as<T>`, `as_string` and `as_wstring`:

To properly handle string concatenation with `utree`, we make use of `as_string[]`. We also use `as<T>` to explicitly create a `utree` symbol node.
```cpp
utree ut;

typedef as<utf8_symbol_type> as_symbol_type;
as_symbol_type const as_symbol = as_symbol_type();

test_parser_attr("foo", as_string[*char_], ut);
std::cout << ut << std::endl; // will output "foo"
BOOST_ASSERT(ut.which() == utree_type::string_type);
ut.clear();

test_parser_attr("foo", as<std::string>()[*char_], ut);
std::cout << ut << std::endl; // will output "foo"
BOOST_ASSERT(ut.which() == utree_type::string_type);
ut.clear();

test_parser_attr("foo", as_symbol[*char_], ut);
std::cout << ut << std::endl; // will output "foo"
BOOST_ASSERT(ut.which() == utree_type::symbol_type);
ut.clear();

test_parser_attr("foo", as<utf8_symbol_type>()[*char_], ut);
std::cout << ut << std::endl; // will output "foo"
BOOST_ASSERT(ut.which() == utree_type::symbol_type);
```

### Nonterminal Parsers

#### Module Headers

```cpp
// forwards to <boost/spirit/home/qi/nonterminal.hpp>
#include <boost/spirit/include/qi_nonterminal.hpp>
```

Also, see Include Structure.

#### Parser Rule

**Description**

The rule is a polymorphic parser that acts as a named placeholder capturing the behavior of a Parsing Expression Grammar expression assigned to it. Naming a Parsing Expression Grammar expression allows it to be referenced later and makes it possible for the rule to call itself. This is one of the most important mechanisms and the reason behind the word "recursive" in recursive descent parsing.

**Header**

```cpp
// forwards to <boost/spirit/home/qi/nonterminal/rule.hpp>
#include <boost/spirit/include/qi_rule.hpp>
```

Also, see Include Structure.

#### Namespace

**Name**

`boost::spirit::qi::rule`
**Synopsis**

```cpp
template <typename Iterator, typename A1, typename A2, typename A3>
struct rule;
```

**Template parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator</td>
<td>The underlying iterator type that the rule is expected to work on.</td>
<td>none</td>
</tr>
<tr>
<td>A1, A2, A3</td>
<td>Either Signature, Skipper or Locals in any order. See table below.</td>
<td>See table below.</td>
</tr>
</tbody>
</table>

Here is more information about the template parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>Specifies the rule's synthesized (return value) and inherited attributes (arguments). More on this here: <a href="#">Nonterminal</a>.</td>
<td>unused_type. When Signature defaults to unused_type, the effect is the same as specifying a signature of <code>void()</code> which is also equivalent to unused_type()</td>
</tr>
<tr>
<td>Skipper</td>
<td>Specifies the rule's skipper parser. Specify this if you want the rule to skip white spaces. If this is not specified, the rule is an &quot;implicit lexeme&quot; and will not skip spaces. &quot;implicit lexeme&quot; rules can still be called with a skipper. In such a case, the rule does a pre-skip just as all lexemes and primitives do.</td>
<td>unused_type</td>
</tr>
<tr>
<td>Locals</td>
<td>Specifies the rule's local variables. See <a href="#">Nonterminal</a>.</td>
<td>unused_type</td>
</tr>
</tbody>
</table>

**Model of**

`Nonterminal`

**Notation**

- `r, r2` Rules
- `p` A parser expression
- `Iterator` The underlying iterator type that the rule is expected to work on.
- `A1, A2, A3` Either Signature, Skipper or Locals in any order.

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in `Nonterminal`. 
### Expression

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rule&lt;Iterator, A1, A2, A3&gt;</code></td>
<td>Rule declaration. <em>Iterator</em> is required. <em>A1, A2, A3</em> are optional and can be specified in any order. <em>name</em> is an optional string that gives the rule its name, useful for debugging and error handling.</td>
</tr>
<tr>
<td><code>r(name);</code></td>
<td>Copy construct rule <em>r</em> from rule <em>r2</em>.</td>
</tr>
<tr>
<td><code>r = r2;</code></td>
<td>Assign rule <em>r2</em> to <em>r</em>.</td>
</tr>
<tr>
<td><code>r.alias()</code></td>
<td>return an alias of <em>r</em>. The alias is a parser that holds a reference to <em>r</em>.</td>
</tr>
<tr>
<td><code>r.copy()</code></td>
<td>Get a copy of <em>r</em>.</td>
</tr>
<tr>
<td><code>r = p;</code></td>
<td>Rule definition. This is equivalent to <em>r %= p</em> (see below) if there are no semantic actions attached anywhere in <em>p</em>.</td>
</tr>
<tr>
<td><code>r %= p;</code></td>
<td>Auto-rule definition. The attribute of <em>p</em> should be compatible with the synthesized attribute of <em>r</em>. When <em>p</em> is successful, its attribute is automatically propagated to <em>r</em>’s synthesized attribute.</td>
</tr>
<tr>
<td><code>r.name()</code></td>
<td>Retrieve the current name of the rule object.</td>
</tr>
<tr>
<td><code>r.name(name)</code></td>
<td>Set the current name of the rule object to be <em>name</em>.</td>
</tr>
</tbody>
</table>

### Attributes

The parser attribute of the rule is `T`, its synthesized attribute. See [Attribute](#).

### Complexity

The complexity is defined by the complexity of the RHS parser, *p*.

### Example

**Note**

The test harness for the example(s) below is presented in the [Basics Examples](#) section.

Some using declarations:

```cpp
guard using boost::spirit::qi::rule;  
guard using boost::spirit::qi::int_;    
guard using boost::spirit::qi::locals;  
guard using boost::spirit::qi::_1;      
guard using boost::spirit::qi::_a;      
guard using boost::spirit::ascii::alpha; 
guard using boost::spirit::ascii::char_; 
guard using boost::spirit::ascii::space_type;
```

Basic rule:
Rule with synthesized attribute:

```cpp
rule<char const*, int()> ra;
ra = int_;  
int i;
test_parser_attr("123", ra, i);
assert(i == 123);
```

Rule with skipper and synthesized attribute:

```cpp
rule<char const*, std::vector<int>(), space_type> rs;
rs = *int_;  
std::vector<int> v;
test_phrase_parser_attr("123 456 789", rs, v);
assert(v[0] == 123);
assert(v[1] == 456);
assert(v[2] == 789);
```

Rule with one local variable:

```cpp
rule<char const*, locals<char> > rl;
rl = alpha[_a = _1] >> char_(_a); // get two identical characters
  test_parser("aa", rl); // pass
  test_parser("ax", rl); // fail
```

**Parser Grammar**

**Description**

The grammar encapsulates a set of rules (as well as primitive parsers (PrimitiveParser) and sub-grammars). The grammar is the main mechanism for modularization and composition. Grammars can be composed to form more complex grammars.

**Header**

```cpp
// forwards to <boost/spirit/home/qi/nonterminal/grammar.hpp>
#include <boost/spirit/include/qi_grammar.hpp>
```

Also, see Include Structure.

**Namespace**

**Name**

boost::spirit::qi::grammar
**Synopsis**

```cpp
template <typename Iterator, typename A1, typename A2, typename A3>
struct grammar;
```

**Template parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator</td>
<td>The underlying iterator type that the rule is expected to work on.</td>
<td>none</td>
</tr>
<tr>
<td>A1, A2, A3</td>
<td>Either Signature, Skipper or Locals in any order. See table below.</td>
<td>See table below.</td>
</tr>
</tbody>
</table>

Here is more information about the template parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>Specifies the grammar's synthesized (return value) and inherited attributes (arguments). More on this here: Nonterminal.</td>
<td>unused_type. When Signature defaults to unused_type, the effect is the same as specifying a signature of void() which is also equivalent to unused_type()</td>
</tr>
<tr>
<td>Skipper</td>
<td>Specifies the grammar's skipper parser. Specify this if you want the grammar to skip white spaces.</td>
<td>unused_type</td>
</tr>
<tr>
<td>Locals</td>
<td>Specifies the grammar's local variables. See Nonterminal.</td>
<td>unused_type</td>
</tr>
</tbody>
</table>

**Model of**

Nonterminal

**Notation**

g  A grammar

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in Nonterminal.
Grammar definition. name is an optional string that gives the grammar its name, useful for debugging and error handling.

```
template <typename Iterator>
struct my_grammar : grammar<Iterator, or, A1, A2, A3>
{
  my_grammar() : my_grammar()
  {
    // Rule definitions
    start = /* ... */;
  }

  rule<Iterator, A1, A2, A3> start;
  // more rule declarations...
};
```

**Note**

The template parameters of a grammar and its start rule (the rule passed to the grammar's base class constructor) must match, otherwise you will see compilation errors.

**Attributes**

The parser attribute of the grammar is T, its synthesized attribute. See Attribute

**Complexity**

The complexity is defined by the complexity of the its definition.

**Example**

```
using boost::spirit::ascii::space_type;
using boost::spirit::int_;
using boost::spirit::qi::grammar;
using boost::spirit::qi::rule;
```

Basic grammar usage:

---

**Spirit 2.5**

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/
```cpp
struct num_list : grammar<char const*, space_type>
{
    num_list() : base_type(start)
    {
        using boost::spirit::int_
        num = int_
        start = num >> *(',') >> num;
    }
    rule<char const*, space_type> start, num;
};
```

How to use the example grammar:

```cpp
num_list nlist;
test_phrase_parser("123, 456, 789", nlist);
```

**Numeric Parsers**

The library includes a couple of predefined objects for parsing signed and unsigned integers and real numbers. These parsers are fully parametric. Most of the important aspects of numeric parsing can be finely adjusted to suit. This includes the radix base, the minimum and maximum number of allowable digits, the exponent, the fraction etc. Policies control the real number parsers' behavior. There are some predefined policies covering the most common real number formats but the user can supply her own when needed.

The numeric parsers are fine tuned (employing loop unrolling and extensive template metaprogramming) with exceptional performance that rivals the low level C functions such as `atof`, `strtod`, `atol`, `strtol`. Benchmarks reveal up to 4X speed over the C counterparts. This goes to show that you can write extremely tight generic C++ code that rivals, if not surpasses C.

**Module Header**

```cpp
// forwards to <boost/spirit/home/qi/numeric.hpp>
#include <boost/spirit/include/qi_numeric.hpp>
```

Also, see Include Structure.

**Unsigned Integer Parsers (uint_, etc.)**

**Description**

The `uint_parser` class is the simplest among the members of the numerics package. The `uint_parser` can parse unsigned integers of arbitrary length and size. The `uint_parser` parser can be used to parse ordinary primitive C/C++ integers or even user defined scalars such as bigints (unlimited precision integers) as long as the type follows certain expression requirements (documented below). The `uint_parser` is a template class. Template parameters fine tune its behavior.

**Header**

```cpp
// forwards to <boost/spirit/home/qi/numeric/uint.hpp>
#include <boost/spirit/include/qi_uint.hpp>
```

Also, see Include Structure.
Namespace

Name

boost::spirit::lit // alias: boost::spirit::qi::lit
boost::spirit::bin // alias: boost::spirit::qi::bin
boost::spirit::oct // alias: boost::spirit::qi::oct
boost::spirit::hex // alias: boost::spirit::qi::hex
boost::spirit::ushort_ // alias: boost::spirit::qi::ushort_
boost::spirit::ulong_ // alias: boost::spirit::qi::ulong_
boost::spirit::uint_ // alias: boost::spirit::qi::uint_
boost::spirit::ulong_long // alias: boost::spirit::qi::ulong_long

Note

ulong_long is only available on platforms where the preprocessor constant BOOST_HAS_LONG_LONG is defined (i.e. on platforms having native support for unsigned long long (64 bit) unsigned integer types).

Note

lit is reused by the Character Parsers, and the Numeric Parsers. In general, a char parser is created when you pass in a character, and a numeric parser is created when you use a numeric literal.

Synopsis

```cpp
template <
    typename T,
    unsigned Radix,
    unsigned MinDigits,
    int MaxDigits>
struct uint_parser;
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>The numeric base type of the numeric parser.</td>
<td>none</td>
</tr>
<tr>
<td>Radix</td>
<td>The radix base. This can be any base from 2, 10 or 16</td>
<td>10</td>
</tr>
<tr>
<td>MinDigits</td>
<td>The minimum number of digits allowable.</td>
<td>1</td>
</tr>
<tr>
<td>MaxDigits</td>
<td>The maximum number of digits allowable. If this is -1, then the maximum limit becomes unbounded.</td>
<td>-1</td>
</tr>
</tbody>
</table>
**Model of**

**PrimitiveParser**

**Notation**

\[ n \quad \text{An object of } T, \text{ the numeric base type.} \]

\[ \text{num} \quad \text{Numeric literal, any unsigned integer value, or a Lazy Argument that evaluates to a unsigned integer value.} \]

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in **PrimitiveParser**.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ushort_</code></td>
<td>Parse an unsigned integer using the default radix (10).</td>
</tr>
<tr>
<td><code>uint_</code></td>
<td>Parse an unsigned integer using the default radix (10).</td>
</tr>
<tr>
<td><code>ulong_</code></td>
<td>Parse an unsigned integer using the default radix (10).</td>
</tr>
<tr>
<td><code>ulong_long</code></td>
<td>Parse an unsigned integer using the default radix (10).</td>
</tr>
<tr>
<td><code>lit(num)</code></td>
<td>Match the literal <code>num</code> using the default radix (10). The parser will fail if the parsed value is not equal to the specified value.</td>
</tr>
<tr>
<td><code>ushort_(num)</code></td>
<td>Match the literal <code>num</code> using the default radix (10). The parser will fail if the parsed value is not equal to the specified value.</td>
</tr>
<tr>
<td><code>uint_(num)</code></td>
<td>Match the literal <code>num</code> using the default radix (10). The parser will fail if the parsed value is not equal to the specified value.</td>
</tr>
<tr>
<td><code>ulong_(num)</code></td>
<td>Match the literal <code>num</code> using the default radix (10). The parser will fail if the parsed value is not equal to the specified value.</td>
</tr>
<tr>
<td><code>ulong_long(num)</code></td>
<td>Match the literal <code>num</code> using the default radix (10). The parser will fail if the parsed value is not equal to the specified value.</td>
</tr>
<tr>
<td><code>bin</code></td>
<td>Parse an unsigned integer using radix 2 for <code>bin</code>, radix 8 for <code>oct</code>, and radix 16 for <code>hex</code>.</td>
</tr>
<tr>
<td><code>oct</code></td>
<td>Parse an unsigned integer using radix 2 for <code>bin</code>, radix 8 for <code>oct</code>, and radix 16 for <code>hex</code>.</td>
</tr>
<tr>
<td><code>hex</code></td>
<td>Parse an unsigned integer using radix 2 for <code>bin</code>, radix 8 for <code>oct</code>, and radix 16 for <code>hex</code>.</td>
</tr>
<tr>
<td><code>bin(num)</code></td>
<td>Match the literal <code>num</code> using radix 2 for <code>bin</code>, radix 8 for <code>oct</code>, and radix 16 for <code>hex</code>. The parser will fail if the parsed value is not equal to the specified value.</td>
</tr>
<tr>
<td><code>oct(num)</code></td>
<td>Match the literal <code>num</code> using radix 2 for <code>bin</code>, radix 8 for <code>oct</code>, and radix 16 for <code>hex</code>. The parser will fail if the parsed value is not equal to the specified value.</td>
</tr>
<tr>
<td><code>hex(num)</code></td>
<td>Match the literal <code>num</code> using radix 2 for <code>bin</code>, radix 8 for <code>oct</code>, and radix 16 for <code>hex</code>. The parser will fail if the parsed value is not equal to the specified value.</td>
</tr>
<tr>
<td><code>uint_parser&lt; T, Radix, MinDigits, MaxDigits &gt;()</code></td>
<td>Parse an unsigned integer of type <code>T</code> using radix <code>Radix</code>, with a minimum of <code>MinDigits</code> and a maximum of <code>MaxDigits</code>.</td>
</tr>
<tr>
<td><code>uint_parser&lt; T, Radix, MinDigits, MaxDigits &gt;()(num)</code></td>
<td>Match the literal <code>num</code> of type <code>T</code> using radix <code>Radix</code>, with a minimum of <code>MinDigits</code> and a maximum of <code>MaxDigits</code>. The parser will fail if the parsed value is not equal to the specified value.</td>
</tr>
</tbody>
</table>

**Important**

All numeric parsers check for overflow conditions based on the type `T` the corresponding `uint_parser<>` has been instantiated with. If the parsed number overflows this type the parsing fails. Please be aware that the overflow check is not based on the type of the supplied attribute but solely depends on the template parameter `T`. 
### Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>lit(num)</td>
<td>unused</td>
</tr>
<tr>
<td>ushort_</td>
<td>unsigned short</td>
</tr>
<tr>
<td>ushort_(num)</td>
<td></td>
</tr>
<tr>
<td>uint_</td>
<td>unsigned int</td>
</tr>
<tr>
<td>uint_(num)</td>
<td></td>
</tr>
<tr>
<td>bin</td>
<td></td>
</tr>
<tr>
<td>bin (num)</td>
<td></td>
</tr>
<tr>
<td>oct</td>
<td></td>
</tr>
<tr>
<td>oct (num)</td>
<td></td>
</tr>
<tr>
<td>hex</td>
<td></td>
</tr>
<tr>
<td>hex (num)</td>
<td></td>
</tr>
<tr>
<td>ulong_</td>
<td>unsigned long</td>
</tr>
<tr>
<td>ulong_(num)</td>
<td></td>
</tr>
<tr>
<td>ulong_long</td>
<td>boost::ulong_long_type</td>
</tr>
<tr>
<td>ulong_long (num)</td>
<td></td>
</tr>
<tr>
<td><code>uint_parser&lt;T, Radix, MinDigits, MaxDigits&gt;()(num)</code></td>
<td>T</td>
</tr>
</tbody>
</table>

### Complexity

$O(N)$, where $N$ is the number of digits being parsed.

### Minimum Expression Requirements for $\tau$

For the numeric base type, $\tau$, the expression requirements below must be valid:
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>T()</td>
<td>Default construct.</td>
</tr>
<tr>
<td>T(0)</td>
<td>Construct from an int.</td>
</tr>
<tr>
<td>n + n</td>
<td>Addition.</td>
</tr>
<tr>
<td>n * n</td>
<td>Multiplication.</td>
</tr>
<tr>
<td><code>std::numeric_limits&lt;T&gt;::is_bounded</code></td>
<td>true or false if T bounded.</td>
</tr>
<tr>
<td><code>std::numeric_limits&lt;T&gt;::digits</code></td>
<td>Maximum Digits for T, radix digits. Required only if T is bounded.</td>
</tr>
<tr>
<td><code>std::numeric_limits&lt;T&gt;::digits10</code></td>
<td>Maximum Digits for T, base 10. Required only if T is bounded.</td>
</tr>
<tr>
<td><code>std::numeric_limits&lt;T&gt;::max()</code></td>
<td>Maximum value for T. Required only if T is bounded.</td>
</tr>
<tr>
<td><code>std::numeric_limits&lt;T&gt;::min()</code></td>
<td>Minimum value for T. Required only if T is bounded.</td>
</tr>
</tbody>
</table>

**Example**

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::phoenix::val;
using boost::spirit::qi::lit;
using boost::spirit::qi::uint_;  
using boost::spirit::qi::uint_parser;
```

Basic unsigned integers:

```cpp
// unsigned int
test_parser("12345", uint_);  
test_parser("12345", uint_(12345));
test_parser("12345", uint_(val(12345)));
```

```cpp
// literals
test_parser("12345", lit(12345));
test_parser("12345", lit(val(12345)));
```

Thousand separated number parser:
Signed Integer Parsers (\texttt{int\_}, etc.)

Description

The \texttt{int\_parser} can parse signed integers of arbitrary length and size. This is almost the same as the \texttt{uint\_parser}. The only difference is the additional task of parsing the `+' or `-' sign preceding the number. The class interface is the same as that of the \texttt{uint\_parser}.

The \texttt{int\_parser} parser can be used to parse ordinary primitive C/C++ integers or even user defined scalars such as bigints (unlimited precision integers) as long as the type follows certain expression requirements (documented below).

Header

\begin{quote}
// forwards to <boost/spirit/home/qi/numeric/int.hpp>
#include <boost/spirit/include/qi_int.hpp>
\end{quote}

Also, see \texttt{Include Structure}.

Namespace

\begin{quote}
\begin{verbatim}
boost::spirit::lit // alias: boost::spirit::qi::lit
boost::spirit::short_ // alias: boost::spirit::qi::short_
boost::spirit::int_ // alias: boost::spirit::qi::int_
boost::spirit::long_ // alias: boost::spirit::qi::long_
boost::spirit::long_long // alias: boost::spirit::qi::long_long
\end{verbatim}
\end{quote}

Note

\begin{quote}
\begin{itemize}
\item \texttt{long\_long} is only available on platforms where the preprocessor constant \texttt{BOOST\_HAS\_LONG\_LONG} is defined (i.e. on platforms having native support for \texttt{signed long long} (64 bit) unsigned integer types).
\end{itemize}
\end{quote}

Note

\begin{quote}
\begin{itemize}
\item \texttt{lit} is reused by the \texttt{Character Parsers}, and the \texttt{Numeric Parsers}. In general, a \texttt{char parser} is created when you pass in a character, and a \texttt{numeric parser} is created when you use a numeric literal.
\end{itemize}
\end{quote}
Synopsis

```cpp
template <
    typename T,
    unsigned Radix,
    unsigned MinDigits,
    int MaxDigits>
struct int_parser;
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>The numeric base type of the numeric parser.</td>
<td>none</td>
</tr>
<tr>
<td>Radix</td>
<td>The radix base. This can be any base from 2..10 and 16</td>
<td>10</td>
</tr>
<tr>
<td>MinDigits</td>
<td>The minimum number of digits allowable.</td>
<td>1</td>
</tr>
<tr>
<td>MaxDigits</td>
<td>The maximum number of digits allowable. If this is -1, then the maximum limit becomes unbounded.</td>
<td>-1</td>
</tr>
</tbody>
</table>

Model of

PrimitiveParser

Notation

- `n` An object of T, the numeric base type.
- `num` Numeric literal, any signed integer value, or a Lazy Argument that evaluates to a signed integer value.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveParser.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>short_</code></td>
<td></td>
</tr>
<tr>
<td><code>int_</code></td>
<td></td>
</tr>
<tr>
<td><code>long_</code></td>
<td></td>
</tr>
<tr>
<td><code>long_long</code></td>
<td></td>
</tr>
</tbody>
</table>

Parse a signed integer using the default radix (10).

| lit(num) |
| `short_(num)` |
| `int_(num)` |
| `long_(num)` |
| `long_long(num)` |

Match the literal `num` using the default radix (10). The parser will fail if the parsed value is not equal to the specified value.

| `int_parser<` |
| `T, Radix, MinDigits, MaxDigits` |
| `>()` |

Parse a signed integer of type `T` using radix `Radix`, with a minimum of `MinDigits` and a maximum of `MaxDigits`.

| `int_parser<` |
| `T, Radix, MinDigits, MaxDigits` |
| `>()(num)` |

Match the literal `num` of type `T` using radix `Radix`, with a minimum of `MinDigits` and a maximum of `MaxDigits`. The parser will fail if the parsed value is not equal to the specified value.

---

**Important**

All numeric parsers check for overflow conditions based on the type `T` the corresponding `int_parser<>` has been instantiated with. If the parsed number overflows this type the parsing fails. Please be aware that the overflow check is not based on the type of the supplied attribute but solely depends on the template parameter `T`. 
## Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lit(num)</code></td>
<td>unused</td>
</tr>
<tr>
<td><code>short_</code> <code>&lt;short_&gt;(num)</code></td>
<td>short</td>
</tr>
<tr>
<td><code>int_</code> <code>&lt;int_&gt;(num)</code></td>
<td>int</td>
</tr>
<tr>
<td><code>long_</code> <code>&lt;long_&gt;(num)</code></td>
<td>long</td>
</tr>
<tr>
<td><code>boost::long_long_type</code> <code>&lt;long_long&gt;(num)</code></td>
<td></td>
</tr>
<tr>
<td><code>int_parser&lt;T, Radix, MinDigits, MaxDigits&gt;()</code></td>
<td><code>T</code></td>
</tr>
</tbody>
</table>

## Complexity

$O(N)$, where $N$ is the number of digits being parsed plus the sign.

## Minimum Expression Requirements for $\tau$

For the numeric base type, $\tau$, the expression requirements below must be valid:
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>T()</td>
<td>Default construct.</td>
</tr>
<tr>
<td>T(0)</td>
<td>Construct from an int.</td>
</tr>
<tr>
<td>n + n</td>
<td>Addition.</td>
</tr>
<tr>
<td>n - n</td>
<td>Subtraction.</td>
</tr>
<tr>
<td>n * n</td>
<td>Multiplication.</td>
</tr>
<tr>
<td><code>std::numeric_limits&lt;T&gt;::is_bounded</code></td>
<td>true or false if T bounded.</td>
</tr>
<tr>
<td><code>std::numeric_limits&lt;T&gt;::digits</code></td>
<td>Maximum Digits for T, radix digits. Required only if T is bounded.</td>
</tr>
<tr>
<td><code>std::numeric_limits&lt;T&gt;::digits10</code></td>
<td>Maximum Digits for T, base 10. Required only if T is bounded.</td>
</tr>
<tr>
<td><code>std::numeric_limits&lt;T&gt;::max()</code></td>
<td>Maximum value for T. Required only if T is bounded.</td>
</tr>
<tr>
<td><code>std::numeric_limits&lt;T&gt;::min()</code></td>
<td>Minimum value for T. Required only if T is bounded.</td>
</tr>
</tbody>
</table>

### Example

#### Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::phoenix::val;
using boost::spirit::qi::lit;
using boost::spirit::qi::int_;`
```

Basic signed integers:
// signed int
test_parser("+12345", int_);
test_parser("-12345", int_);
test_parser("+12345", int_(12345));
test_parser("-12345", int_(12345));
test_parser("+12345", int_(val(12345)));
test_parser("-12345", int_(val(-12345)));

// literals
test_parser("+12345", lit(12345));
test_parser("-12345", lit(-12345));
test_parser("+12345", lit(val(12345)));
test_parser("-12345", lit(val(-12345)));

### Real Number Parsers (float_, double_, etc.)

**Description**

The real_parser can parse real numbers of arbitrary length and size limited by its template parameter, T. The numeric base type T can be a user defined numeric type such as fixed_point (fixed point reals) and bignum (unlimited precision numbers) as long as the type follows certain expression requirements (documented below).

**Header**

// forwards to <boost/spirit/home/qi/numeric/real.hpp>
#include <boost/spirit/include/qi_real.hpp>

Also, see Include Structure.

**Namespace**

**Name**

boost::spirit::lit // alias: boost::spirit::qi::lit
boost::spirit::float_ // alias: boost::spirit::qi::float_
boost::spirit::double_ // alias: boost::spirit::qi::double_
boost::spirit::long_double // alias: boost::spirit::qi::long_double

**Note**

lit is reused by the Character Parsers, and the Numeric Parsers. In general, a char parser is created when you pass in a character, and a numeric parser is created when you use a numeric literal.
Synopsis

```cpp
template <typename T, typename RealPolicies>
struct real_parser;
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>The numeric base type of the numeric parser.</td>
<td>none</td>
</tr>
<tr>
<td>RealPolicies</td>
<td>Policies control the parser's behavior.</td>
<td>real_policies&lt;T&gt;</td>
</tr>
</tbody>
</table>

Model of

*PrimitiveParser*

Notation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>An object of T, the numeric base type.</td>
</tr>
<tr>
<td>num</td>
<td>Numeric literal, any real value, or a Lazy Argument that evaluates to a real value.</td>
</tr>
<tr>
<td>RP</td>
<td>A RealPolicies (type).</td>
</tr>
<tr>
<td>exp</td>
<td>A int exponent.</td>
</tr>
<tr>
<td>b</td>
<td>A bool flag.</td>
</tr>
<tr>
<td>f, l</td>
<td>ForwardIterator, first/last iterator pair.</td>
</tr>
</tbody>
</table>

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in *PrimitiveParser*. 
### Expression

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>float_</td>
<td>Parse a real using the default policies (real_policies&lt;T&gt;).</td>
</tr>
<tr>
<td>double_</td>
<td></td>
</tr>
<tr>
<td>long_double</td>
<td></td>
</tr>
<tr>
<td>lit(num)</td>
<td>Match the literal num using the default policies (real_policies&lt;T&gt;).</td>
</tr>
<tr>
<td>float_(num)</td>
<td>The parser will fail if the parsed value is not equal to the specified value.</td>
</tr>
<tr>
<td>double_(num)</td>
<td></td>
</tr>
<tr>
<td>long_double(num)</td>
<td></td>
</tr>
<tr>
<td>real_parser&lt;T, RealPolicies&gt;()</td>
<td>Parse a real of type T using RealPolicies.</td>
</tr>
<tr>
<td>real_parser&lt;T, RealPolicies&gt;()(num)</td>
<td>Match the literal num of type T using RealPolicies. The parser will fail if the parsed value is not equal to the specified value.</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>lit(num)</td>
<td>unused</td>
</tr>
<tr>
<td>float_</td>
<td>float</td>
</tr>
<tr>
<td>float_(num)</td>
<td></td>
</tr>
<tr>
<td>double_</td>
<td>double</td>
</tr>
<tr>
<td>double_(num)</td>
<td></td>
</tr>
<tr>
<td>long_double</td>
<td>long double</td>
</tr>
<tr>
<td>long_double(num)</td>
<td></td>
</tr>
<tr>
<td>real_parser&lt;T, RealPolicies&gt;()</td>
<td></td>
</tr>
<tr>
<td>real_parser&lt;T, RealPolicies&gt;()(num)</td>
<td></td>
</tr>
</tbody>
</table>
**Complexity**

O(N), where N is the number of characters (including the digits, exponent, sign, etc.) being parsed.

**Minimum Expression Requirements for \( \mathcal{T} \)**

The numeric base type, \( \mathcal{T} \), the minimum expression requirements listed below must be valid. Take note that additional requirements may be imposed by custom policies.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{T}() )</td>
<td>Default construct.</td>
</tr>
<tr>
<td>( \mathcal{T}(0) )</td>
<td>Construct from an int.</td>
</tr>
<tr>
<td>( n + n )</td>
<td>Addition.</td>
</tr>
<tr>
<td>( n - n )</td>
<td>Subtraction.</td>
</tr>
<tr>
<td>( n \times n )</td>
<td>Multiplication.</td>
</tr>
<tr>
<td><code>std::numeric_limits&lt;\mathcal{T}&gt;::is_bounded</code></td>
<td>true or false if ( \mathcal{T} ) bounded.</td>
</tr>
<tr>
<td><code>std::numeric_limits&lt;\mathcal{T}&gt;::digits</code></td>
<td>Maximum Digits for ( \mathcal{T} ), radix digits. Required only if ( \mathcal{T} ) is bounded.</td>
</tr>
<tr>
<td><code>std::numeric_limits&lt;\mathcal{T}&gt;::digits10</code></td>
<td>Maximum Digits for ( \mathcal{T} ), base 10. Required only if ( \mathcal{T} ) is bounded.</td>
</tr>
<tr>
<td><code>std::numeric_limits&lt;\mathcal{T}&gt;::max()</code></td>
<td>Maximum value for ( \mathcal{T} ). Required only if ( \mathcal{T} ) is bounded.</td>
</tr>
<tr>
<td><code>std::numeric_limits&lt;\mathcal{T}&gt;::min()</code></td>
<td>Minimum value for ( \mathcal{T} ). Required only if ( \mathcal{T} ) is bounded.</td>
</tr>
<tr>
<td><code>boost::spirit::traits::scale(exp, n)</code></td>
<td>Multiply ( n ) by ( 10^\text{exp} ). Default implementation is provided for float, double and long double.</td>
</tr>
<tr>
<td><code>boost::spirit::traits::negate(b, n)</code></td>
<td>Negate ( n ) if ( b ) is true. Default implementation is provided for float, double and long double.</td>
</tr>
<tr>
<td><code>boost::spirit::traits::is_equal_to_one(n)</code></td>
<td>Return true if ( n ) is equal to 1.0. Default implementation is provided for float, double and long double.</td>
</tr>
</tbody>
</table>

**Note**
The additional spirit real number traits above are provided to allow custom implementations to implement efficient real number parsers. For example, for certain custom real numbers, scaling to a base 10 exponent is a very cheap operation.

**RealPolicies**
The `RealPolicies` template parameter is a class that groups all the policies that control the parser's behavior. Policies control the real number parsers' behavior.

The default is `real_policies<\mathcal{T}>`. The default is provided to take care of the most common case (there are many ways to represent, and hence parse, real numbers). In most cases, the default policies are sufficient and can be used straight out of the box. They are designed to parse C/C++ style floating point numbers of the form `\text{nnn.fff.Eeee}` where `nnn` is the whole number part, `fff` is the fractional part, `E` is `\text{e}` or `\text{E}` and `eee` is the exponent optionally preceded by `-` or `+` with the additional detection of NaN.
and Inf as mandated by the C99 Standard and proposed for inclusion into the C++0x Standard: nan, nan(...), inf and infinity (the matching is case-insensitive). This corresponds to the following grammar:

```cpp
sign = lit('+') | '-'

nan = -lit("1.0#") >> no_case["nan"]
    >> -("'\n' >> *(char_ - ')'), >> ")

inf = no_case[lit("inf") >> -lit("inity")]

floating_literal = -sign >>
    (nan
     | inf
     | fractional_constant >> !exponent_part
     | +digit >> exponent_part)

fractional_constant = *digit >> "." >> +digit
    | +digit >> -lit(".")

exponent_part = (lit('e') | 'E') >> -sign >> +digit
```

There are four `RealPolicies` predefined for immediate use:

### Table 4. Predefined Policies

<table>
<thead>
<tr>
<th>Policies</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unreal_policies&lt;double&gt;</td>
<td>Without sign.</td>
</tr>
<tr>
<td>real_policies&lt;double&gt;</td>
<td>With sign.</td>
</tr>
<tr>
<td>strict_ureal_policies&lt;double&gt;</td>
<td>Without sign, dot required.</td>
</tr>
<tr>
<td>strict_real_policies&lt;double&gt;</td>
<td>With sign, dot required.</td>
</tr>
</tbody>
</table>

**Note**

Integers are considered a subset of real numbers, so for instance, `double_` recognizes integer numbers (without a dot) just as well. To avoid this ambiguity, `strict_ureal_policies` and `strict_real_policies` require a dot to be present for a number to be considered a successful match.

**RealPolicies Expression Requirements**

For models of `RealPolicies` the following expressions must be valid:
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>RP::allow_leading_dot</td>
<td>Allow leading dot.</td>
</tr>
<tr>
<td>RP::allow_trailing_dot</td>
<td>Allow trailing dot.</td>
</tr>
<tr>
<td>RP::expect_dot</td>
<td>Require a dot.</td>
</tr>
<tr>
<td>RP::parse_sign(f, l)</td>
<td>Parse the prefix sign (e.g. <code>.</code>). Return true if successful, otherwise false.</td>
</tr>
<tr>
<td>RP::parse_n(f, l, n)</td>
<td>Parse the integer at the left of the decimal point. Return true if successful, otherwise false. If successful, place the result into n.</td>
</tr>
<tr>
<td>RP::parse_dot(f, l)</td>
<td>Parse the decimal point. Return true if successful, otherwise false.</td>
</tr>
<tr>
<td>RP::parse_frac_n(f, l, n)</td>
<td>Parse the fraction after the decimal point. Return true if successful, otherwise false. If successful, place the result into n.</td>
</tr>
<tr>
<td>RP::parse_exp(f, l)</td>
<td>Parse the exponent prefix (e.g. <code>e</code>). Return true if successful, otherwise false.</td>
</tr>
<tr>
<td>RP::parse_exp_n(f, l, n)</td>
<td>Parse the actual exponent. Return true if successful, otherwise false. If successful, place the result into n.</td>
</tr>
<tr>
<td>RP::parse_nan(f, l, n)</td>
<td>Parse a NaN. Return true if successful, otherwise false. If successful, place the result into n.</td>
</tr>
<tr>
<td>RP::parse_inf(f, l, n)</td>
<td>Parse an Inf. Return true if successful, otherwise false. If successful, place the result into n.</td>
</tr>
</tbody>
</table>

The `parse_nan` and `parse_inf` functions get called whenever:

- a number to parse does not start with a digit (after having successfully parsed an optional sign)

or

- after a real number of the value 1 (having no exponential part and a fractional part value of 0) has been parsed.

The first call recognizes representations of NaN or Inf starting with a non-digit character (such as NaN, Inf, QNaN etc.). The second call recognizes representation formats starting with a 1.0 (such as "1.0#NAN" or "1.0#INF" etc.).

The functions should return true if a Nan or Inf has been found. In this case the attribute n should be set to the matched value (NaN or Inf). The optional sign will be automatically applied afterwards.

**RealPolicies Specializations**

The easiest way to implement a proper real parsing policy is to derive a new type from the the type `real_policies` while overriding the aspects of the parsing which need to be changed. For example, here's the implementation of the predefined `strict_real_policies`:
template <typename T>
struct strict_real_policies : real_policies<T> {
    static bool const expect_dot = true;
};

**Example**

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::phoenix::val;
using boost::spirit::qi::double_;
using boost::spirit::qi::real_parser;
using boost::spirit::qi::lit;
```

Basic real number parsing:

```cpp
// double
test_parser("+12345e6", double_);
test_parser("-12345e6", double_);
test_parser("+12345e6", double_(12345e6));
test_parser("-12345e6", double_(-123456e6));
test_parser("+12345e6", double_(val(12345e6)));
test_parser("-12345e6", double_(val(-123456e6)));

// literals
test_parser("+12345e6", lit(12345e6));
test_parser("-12345e6", lit(-123456e6));
test_parser("+12345e6", lit(val(12345e6)));
test_parser("-12345e6", lit(val(-123456e6)));
```

A custom real number policy:
// These policies can be used to parse thousand separated numbers with at most 2 decimal digits after the decimal point. e.g. 123,456,789.01

template <typename T>
struct ts_real_policies : boost::spirit::qi::ureal_policies<T>
{
    // 2 decimal places Max
    template <typename Iterator, typename Attribute>
    static bool
    parse_frac_n(Iterator& first, Iterator const& last, Attribute& attr)
    {
        return boost::spirit::qi::extract_uint<T, 10, 1, 2, true>::call(first, last, attr);
    }

    // No exponent
    template <typename Iterator>
    static bool
    parse_exp(Iterator&, Iterator const&)
    {
        return false;
    }

    // No exponent
    template <typename Iterator, typename Attribute>
    static bool
    parse_exp_n(Iterator&, Iterator const&, Attribute&)
    {
        return false;
    }

    // Thousands separated numbers
    template <typename Iterator, typename Attribute>
    static bool
    parse_n(Iterator& first, Iterator const& last, Attribute& attr)
    {
        using boost::spirit::qi::uint_parser;
        namespace qi = boost::spirit::qi;

        uint_parser<unsigned, 10, 1, 3> uint3;
        uint_parser<unsigned, 10, 3, 3> uint3_3;

        T result = 0;
        if (parse(first, last, uint3, result))
        {
            bool hit = false;
            T n;
            Iterator save = first;

            while (qi::parse(first, last, ',') && qi::parse(first, last, uint3_3, n))
            {
                result = result * 1000 + n;
                save = first;
                hit = true;
            }

            first = save;
            if (hit)
        }
    }
};
And its use:

```cpp
test_parser("123,456,789.01", ts_real); 
test_parser("123,456,789.01", ts_real(123456789.01));
```

### Boolean Parser (bool_)

#### Description

The `bool_` parser can parse booleans of arbitrary type, `B`. The boolean base type `T` can be a user defined boolean type as long as the type follows certain expression requirements (documented below).

#### Header

```cpp
#include <boost/spirit/include/qi_bool.hpp>
```

Also, see Include Structure.

#### Namespace

- `boost::spirit::bool_` // alias: `boost::spirit::qi::bool_`
- `boost::spirit::true_` // alias: `boost::spirit::qi::true_`
- `boost::spirit::false_` // alias: `boost::spirit::qi::false_`

#### Synopsis

```cpp
template <typename T, typename BooleanPolicies> 
struct bool_parser;
```

#### Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>B</code></td>
<td>The boolean type of the boolean parser.</td>
<td>bool</td>
</tr>
<tr>
<td><code>BooleanPolicies</code></td>
<td>Policies control the parser's behavior.</td>
<td><code>bool_policy&lt;B&gt;</code></td>
</tr>
</tbody>
</table>

#### Model of

*PrimitiveParser*
Notation

BP  A boolean Policies (type).
b  An object of B, the numeric base type.
boolean  Numeric literal, any boolean value, or a Lazy Argument that evaluates to a boolean value.
f, l  ForwardIterator. first/last iterator pair.
attr  An attribute value.

Context  The type of the parse context of the current invocation of the bool_ parser.
ctx  An instance of the parse context, Context.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool_</td>
<td>Parse a boolean using the default policies (bool_policies&lt;T&gt;).</td>
</tr>
</tbody>
</table>
| lit(boolean)  
bool_(boolean) | Match the literal boolean using the default policies (bool_policies<T>). The parser will fail if the parsed value is not equal to the specified value. |
| true_       
false_      | Match "true" and "false", respectively. |
| bool_parser<T, BoolPolicies>(()) | Parse a real of type T using BoolPolicies. |
| bool_parser<T, BoolPolicies>(boolean) | Match the literal boolean of type T using BoolPolicies. The parser will fail if the parsed value is not equal to the specified value. |

Note

All boolean parsers properly respect the no_case[] directive.
**Attributes**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lit(boolean)</code></td>
<td>unused</td>
</tr>
<tr>
<td><code>true_</code></td>
<td>bool</td>
</tr>
<tr>
<td><code>false_</code></td>
<td></td>
</tr>
<tr>
<td><code>bool_</code></td>
<td></td>
</tr>
<tr>
<td><code>bool_(boolean)</code></td>
<td></td>
</tr>
<tr>
<td><code>bool_parser&lt;T, BoolPolicies&gt;()</code></td>
<td><code>T</code></td>
</tr>
</tbody>
</table>

**Complexity**

O(N), where N is the number of characters being parsed.

**Minimum Expression Requirements for** \(B\)

The boolean type, \(B\), the minimum expression requirements listed below must be valid. Take note that additional requirements may be imposed by custom policies.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B(bool))</td>
<td>Constructible from a <code>bool</code>.</td>
</tr>
</tbody>
</table>

**Boolean Policies**

The boolean Policies template parameter is a class that groups all the policies that control the parser\'s behavior. Policies control the boolean parsers\' behavior.

The default is `bool_policies<bool>`. The default is provided to take care of the most common case (there are many ways to represent, and hence parse, boolean numbers). In most cases, the default policies are sufficient and can be used straight out of the box. They are designed to parse boolean value of the form "true" and "false".

**Boolean Policies Expression Requirements**

For models of boolean Policies the following expressions must be valid:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BP::parse_true(f, l, attr, ctx)</code></td>
<td>Parse a true value.</td>
</tr>
<tr>
<td><code>BP::parse_false(f, l, attr, ctx)</code></td>
<td>Parse a false value.</td>
</tr>
</tbody>
</table>

The functions should return true if the required representations of `true` or `false` have been found. In this case the attribute \(n\) should be set to the matched value (`true` or `false`).
Boolean Policies Specializations

The easiest way to implement a proper boolean parsing policy is to derive a new type from the the type bool_policies while overriding the aspects of the parsing which need to be changed. For example, here's the implementation of a boolean parsing policy interpreting the string "eurt" (i.e. "true" spelled backwards) as false:

```
struct backwards_bool_policies : qi::bool_policies<> {
    // we want to interpret a 'true' spelled backwards as 'false'
    template <typename Iterator, typename Attribute, typename Context>
    static bool parse_false(Iterator& first, Iterator const& last, Attribute& attr, Context& ctx) {
        namespace qi = boost::spirit::qi;
        if (qi::detail::string_parse("eurt", first, last, qi::unused, qi::unused)) {
            spirit::traits::assign_to(false, attr, ctx); // result is false
            return true;
        }
        return false;
    }
};
```

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```
using boost::phoenix::val;
using boost::spirit::qi::bool_;
using boost::spirit::qi::bool_parser;
using boost::spirit::qi::lit;
```

Basic real number parsing:

```
// bool
test_parser("true", bool_);
test_parser("false", bool_);
test_parser("true", bool_(true));
test_parser("false", bool_(false));

test_parser("true", bool_(val(true)));

test_parser("false", bool_(val(true)));

test_parser("false", bool_(val(false)));

// literals

test_parser("true", lit(true));

test_parser("false", lit(false));

test_parser("true", lit(val(true)));

test_parser("false", lit(val(false)));
```

A custom real number policy:
These policies can be used to parse "eurt" (i.e. "true" spelled backwards) as 'false'.

```cpp
struct backwards_bool_policies : boost::spirit::qi::bool_policies<> {
    template <typename Iterator, typename Attribute>
    static bool parse_false(Iterator& first, Iterator const& last, Attribute& attr) {
        namespace qi = boost::spirit::qi;
        if (qi::detail::string_parse("eurt", first, last, qi::unused)) {
            namespace traits = boost::spirit::traits;
            traits::assign_to(false, attr); // result is false
            return true;
        }
        return false;
    }
};
```

And its use:

```cpp
bool_parser<bool, backwards_bool_policies> backwards_bool;

test_parser("true", backwards_bool);
test_parser("eurt", backwards_bool);
test_parser("true", backwards_bool(true));
test_parser("eurt", backwards_bool(false));
```

## Parser Operators

Operators are used as a means for object composition and embedding. Simple parsers may be composed to form composites through operator overloading, crafted to approximate the syntax of Parsing Expression Grammar (PEG). An expression such as:

```
a | b
```

yields a new parser type which is a composite of its operands, a and b.

This module includes different parsers which get instantiated if one of the overloaded operators is used with more primitive parser constructs. It includes Alternative (\(\mid\)), And-predicate (unary \&), Difference (\(-\)), Expect (\(>\)), Kleene star (unary \(*\)), Lists (\(\%\)), Not-predicate (!), Optional (unary \(-\)), Permutation (^), Plus (unary +), Sequence (\(>>\)), and Sequential-Or (\(\mid\mid\)).

### Module Header

```cpp
// forwards to <boost/spirit/home/qi/operator.hpp>
#include <boost/spirit/include/qi/operator.hpp>
```

Also, see Include Structure.

### Alternative Parser (\(a \mid b\))

#### Description

The alternative operator, \(a \mid b\), matches one of two or more operands (\(a, b, \ldots \) etc.):
Alternative operands are tried one by one on a first-match-wins basis starting from the leftmost operand. After a successfully matched alternative is found, the parser concludes its search, essentially short-circuiting the search for other potentially viable candidates. This short-circuiting implicitly gives the highest priority to the leftmost alternative.

Short-circuiting is done in the same manner as C or C++’s logical expressions; e.g. if (x < 3 || y < 2) where, if x < 3, the y < 2 test is not done at all. In addition to providing an implicit priority rule for alternatives which is necessary, given its non-deterministic nature, short-circuiting improves the execution time. If the order of your alternatives is logically irrelevant, strive to put the (expected) most common choice first for maximum efficiency.

**Header**

```cpp
// forwards to <boost/spirit/home/qi/operator/alternative.hpp>
#include <boost/spirit/include/qi_alternative.hpp>
```

Also, see Include Structure.

**Model of**

_NaryParser_

**Notation**

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in _NaryParser_.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

**Attributes**

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
</table>

**Note**

Alternative parsers do not roll back changes made to the outer attribute because of a failed alternative. If you need to enforce that only the succeeded alternative changes the outer attribute please utilize the directive `hold[]`. 
Complexity

The overall complexity of the alternative parser is defined by the sum of the complexities of its elements. The complexity of the alternative parser itself is \( O(N) \), where \( N \) is the number of alternatives.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::ascii::string;
using boost::spirit::qi::int_;
using boost::spirit::qi::_1;
using boost::variant;
```

Simple usage:

```cpp
test_parser("Hello", string("Hello") | int_);
test_parser("123", string("Hello") | int_);
```

Extracting the attribute variant (using Boost.Variant):

```cpp
variant<std::string, int> attr;
test_parser_attr("Hello", string("Hello") | int_, attr);
```

This should print "Hello". Note: There are better ways to extract the value from the variant. See Boost.Variant visitation. This code is solely for demonstration.

```cpp
if (boost::get<int>(&attr))
    std::cout << boost::get<int>(attr) << std::endl;
else
    std::cout << boost::get<std::string>(attr) << std::endl;
```

Extracting the attributes using Semantic Actions with Phoenix (this should print 123):

```cpp
test_parser("123", (string("Hello") | int_)[std::cout << _1 << std::endl]);
```

And-Predicate Parser (\&a)

Description

Syntactic predicates assert a certain conditional syntax to be satisfied before evaluating another production. Similar to semantic predicates, \( \text{eps} \), syntactic predicates do not consume any input. The \text{and-predicate}, \&a, is a positive syntactic predicate that returns a zero length match only if its predicate matches.

Header

```cpp
// forwards to <boost/spirit/home/qi/operator/and_predicate.hpp>
#include <boost/spirit/include/qi_and_predicate.hpp>
```

Also, see Include Structure.
Model of

UnaryParser

Notation

\( a \)

Parser

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;a</td>
<td>If the predicate ( a ) matches, return a zero length match. Otherwise, fail.</td>
</tr>
</tbody>
</table>

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;a</td>
<td>unused_type</td>
</tr>
</tbody>
</table>

Complexity

The complexity is defined by the complexity of the predicate, \( a \)

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::lit;
```

Basic look-ahead example: make sure that the last character is a semicolon, but don't consume it, just peek at the next character:

```cpp
test_phrase_parser("Hello ;", lit("Hello") >> \&lit(';'), false);
```

Difference Parser (\( a - b \))

Description

The difference operator, \( a - b \), is a binary operator that matches the first (LHS) operand but not the second (RHS). ⁸

⁸ Unlike classic Spirit, with Spirit2, the expression will always fail if the RHS is a successful match regardless if the RHS matches less characters. For example, the rule `lit("policeman") - "police"` will always fail to match. Spirit2 does not count the matching chars while parsing and there is no reliable and fast way to check if the LHS matches more than the RHS.
Header

// forwards to <boost/spirit/home/qi/operator/difference.hpp>
#include <boost/spirit/include/qi_difference.hpp>

Also, see Include Structure.

Model of

BinaryParser

Notation

a, b A Parser

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in BinaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>a - b</td>
<td>Parse a but not b.</td>
</tr>
</tbody>
</table>

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>a - b</td>
<td>a: A, b: B --&gt; (a - b): A</td>
</tr>
<tr>
<td></td>
<td>a: Unused, b: B --&gt; (a - b): Unused</td>
</tr>
</tbody>
</table>

Complexity

The complexity of the difference parser is defined by the sum of the complexities of both operands.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

using boost::spirit::ascii::char_;
Expectation Parser ($a > b$)

Description

Like the Sequence, the expectation operator, $a > b$, parses two or more operands ($a, b, ...$), in sequence:

$$a > b > ...$$

However, while the plain Sequence simply returns a no-match (returns false) when one of the elements fail, the expectation: $>$ operator throws an expectation_failure<Iter> when the second or succeeding operands (all operands except the first) fail to match.

Header

```cpp
#include <boost/spirit/include/qi_expect.hpp>
```

Also, see Include Structure.

Model of

NaryParser

Notation

$a,b$ A Parser

Iter A ForwardIterator type

Expectation Failure

When any operand, except the first, fail to match an expectation_failure<Iter> is thrown:

```cpp
template <typename Iter>
struct expectation_failure : std::runtime_error {
    Iter first; // [first, last) iterator pointing
    Iter last;   // to the error position in the input.
    info what_;  // Information about the nature of the error.
};
```

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in NaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a &gt; b$</td>
<td>Match $a$ followed by $b$. If $a$ fails, no-match. If $b$ fails, throw an expectation_failure&lt;Iter&gt;</td>
</tr>
</tbody>
</table>

Attributes

See Compound Attribute Notation.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
</table>
| a > b | a: A, b: B --> (a > b): tuple\<A, B\>  
|          | a: A, b: Unused --> (a > b): A  
|          | a: Unused, b: B --> (a > b): B  
|          | a: Unused, b: Unused --> (a > b): Unused  
|          | a: A, b: A --> (a > b): vector\<A\>  
|          | a: vector\<A\>, b: A --> (a > b): vector\<A\>  
|          | a: A, b: vector\<A\> --> (a > b): vector\<A\>  
|          | a: vector\<A\>, b: vector\<A\> --> (a > b): vector\<A\> |

### Complexity

The overall complexity of the expectation parser is defined by the sum of the complexities of its elements. The complexity of the expectation operator itself is \(O(N)\), where \(N\) is the number of elements in the sequence.

### Example

#### Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::ascii::char_;  
using boost::spirit::qi::expectation_failure;
```

The code below uses an expectation operator to throw an \texttt{expectation\_failure} with a deliberate parsing error when "o" is expected and "i" is what is found in the input. The \texttt{catch} block prints the information related to the error. Note: This is low level code that demonstrates the \textit{bare-metal}. Typically, you use an Error Handler to deal with the error.

```cpp
try  
{  
test_parser("xi", char\_\_('x') > char\_\_('o')); // should throw an exception  
}  
catch (expectation\_failure<char const*> const\& x)  
{  
std::cout << "expected: "; print\_info(x.what\_\_);  
std::cout << "got: \" << std::string(x.first, x.last) << '\"' << std::endl;  
}
```

The code above will print:

```
expected: tag: literal\-char, value: o
  got: "i"
```

### Kleene Parser (\texttt{\*a})

#### Description

The kleene operator, \texttt{\*a}, is a unary operator that matches its operand zero or more times.
Header

// forwards to <boost/spirit/home/qi/operator/kleene.hpp>
#include <boost/spirit/include/qi_kleene.hpp>

Also, see Include Structure.

Model of

    UnaryParser

Notation

  a  A Parser

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>*a</td>
<td>Match a zero or more times.</td>
</tr>
</tbody>
</table>

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
</table>
| *a         | a: A --> *a: vector<A>  
            | a: Unused --> *a: Unused                     |

Complexity

The overall complexity of the Kleene star is defined by the complexity of its subject, a, multiplied by the number of repetitions. The complexity of the Kleene star itself is O(N), where N is the number successful repetitions.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::qi::int_;  
```

Parse a comma separated list of numbers and put them in a vector:
List Parser \((a \% b)\)

Description

The list operator, \(a \% b\), is a binary operator that matches a list of one or more repetitions of \(a\) separated by occurrences of \(b\). This is equivalent to \(a >> *(b >> a)\).

Header

```cpp
// forwards to <boost/spirit/home/qi/operator/list.hpp>
#include <boost/spirit/include/qi_list.hpp>
```

Also, see Include Structure.

Model of

BinaryParser

Notation

\(a, b \quad A\) Parser

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in BinaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a % b)</td>
<td>Match a list of one or more repetitions of (a) separated by occurrences of (b). This is equivalent to (a &gt;&gt; *(b &gt;&gt; a)).</td>
</tr>
</tbody>
</table>

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a % b)</td>
<td>(a: A, b: B \rightarrow (a % b): vector&lt;A&gt;) &lt;br&gt;(a: Unused, b: B \rightarrow (a % b): Unused)</td>
</tr>
</tbody>
</table>

Complexity

The overall complexity of the List is defined by the complexity of its subject, \(a\), multiplied by the number of repetitions. The complexity of the List itself is \(O(N)\), where \(N\) is the number successful repetitions.
Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::qi::int_;  
```

Parse a comma separated list of numbers and put them in a vector:

```cpp
std::vector<int> attr;

// Function to test phrase parser

test_phrase_parser_attr(  
    "111, 222, 333, 444, 555", int_ % ',', attr);

// Output

std::cout  
    << attr[0] << ','  
    << attr[1] << ','  
    << attr[2] << ','  
    << attr[3] << ','  
    << attr[4]  
    << std::endl;
```

**Not-Predicate Parser (\(!a\)**

**Description**

Syntactic predicates assert a certain conditional syntax to be satisfied before evaluating another production. Similar to semantic predicates, `eps`, syntactic predicates do not consume any input. The *not-predicate*, `!a`, is a negative syntactic predicate that returns a zero length match only if its predicate fails to match.

**Header**

```cpp
#include <boost/spirit/include/qi_not_predicate.hpp>
```

Also, see Include Structure.

**Model of**

UnaryParser

**Notation**

\( a \quad A \quad \text{Parser} \)

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in UnaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!a</code></td>
<td>If the predicate <code>a</code> matches, fail. Otherwise, return a zero length match.</td>
</tr>
</tbody>
</table>

**Attributes**

See Compound Attribute Notation.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>!a</td>
<td>unused_type</td>
</tr>
</tbody>
</table>

### Complexity

The complexity is defined by the complexity of the predicate, `a`

### Example

#### Note

The test harness for the example(s) below is presented in the [Basics Examples](#) section.

Some using declarations:

```cpp
using boost::spirit::ascii::char_;  
using boost::spirit::ascii::alpha;  
using boost::spirit::qi::lit;  
using boost::spirit::qi::symbols;  
```

Here's an alternative to the `*(r - x) >> x` idiom using the not-predicate instead. This parses a list of characters terminated by a `;`:

```cpp
test_parser("abcdef;", *(!lit(';') >> char_) >> ';');
```

The following parser ensures that we match distinct keywords (stored in a symbol table). To do this, we make sure that the keyword does not follow an alpha or an underscore:

```cpp
symbols<char, int> keywords;  
keywords = "begin", "end", "for";
```

// This should fail:  
test_parser("beginner", keywords >> !(alpha | '_'));

// This is ok:  
test_parser("end ", keywords >> !(alpha | '_'), false);

// This is ok:  
test_parser("for()", keywords >> !(alpha | '_'), false);

### Optional Parser (`-a`)

#### Description

The optional operator, `-a`, is a unary operator that matches its operand zero or one time.

#### Header

```cpp
// forwards to <boost/spirit/home/qi/operator/optional.hpp>  
#include <boost/spirit/include/qi_optional.hpp>
```

Also, see [Include Structure](#).
Model of

UnaryParser

Notation

a A Parser

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>Match a zero or one time.</td>
</tr>
</tbody>
</table>

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
</table>
| -a         | a: A --> -a: optional<A>  
|            | a: Unused --> -a: Unused |

Complexity

The complexity is defined by the complexity of the operand, a

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::ascii::char_;
using boost::spirit::qi::lexeme;
using boost::spirit::qi::int_;
using boost::fusion::vector;
using boost::fusion::at_c;
using boost::optional;
```

Parse a person info with name (in quotes) optional age and optional sex, all separated by comma.

---

\(^9\) James Bond is shy about his age :-(

Permutation Parser (a ^ b)

Description

The permutation operator, a ^ b, matches one or more operands (a, b, ... etc.) in any order:

a ^ b ^ ...

The operands are the elements in the permutation set. Each element in the permutation set may occur at most once, but not all elements of the given set need to be present. Note that by this definition, the permutation operator is not limited to strict permutations.

For example:

char_('a') ^ 'b' ^ 'c'

matches:

"a", "ab", "abc", "cba", "bca" ... etc.

Header

// forwards to <boost/spirit/home/qi/operator/permutation.hpp>
#include <boost/spirit/include/qi_permutation.hpp>

Also, see Include Structure.

Model of

NaryParser

Notation

a, b A Parser

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in NaryParser.
Expression | Semantics
---|---
a ^ b | Match a or b in any order. Each operand may match zero or one time as long as at least one operand matches.

Attributes

See Compound Attribute Notation.

Expression | Attribute
---|---
a ^ b | a: A, b: B --> (a ^ b): tuple<optional<A>, optional<B>>
a: A, b: Unused --> (a ^ b): optional<A>
a: Unused, b: B --> (a ^ b): optional<B>
a: Unused, b: Unused --> (a ^ b): Unused

Complexity

The overall complexity of the permutation parser is defined by the sum of the complexities of its elements, s, multiplied by log s. The complexity of the permutation parser itself is O(N log N), where N is the number of elements.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::ascii::char_;```

Parse a string containing DNA codes (ACTG)

```cpp
test_parser("ACTGGCTAGACT", *(char_('A') ^ 'C' ^ 'T' ^ 'G'));```

Plus Parser (+a)

Description

The plus operator, +a, is a unary operator that matches its operand one or more times.

Header

```cpp
// forwards to <boost/spirit/home/qi/operator/plus.hpp>
#include <boost/spirit/include/qi_plus.hpp>
```

Also, see Include Structure.

Model of

UnaryParser
Notation

 Parser

 Expression Semantics

 Semantics of an expression is defined only where it differs from, or is not defined in UnaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>+a</td>
<td>Match a one or more times.</td>
</tr>
</tbody>
</table>

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>+a</td>
<td>a: A --&gt; +a: vector&lt;A&gt;</td>
</tr>
<tr>
<td></td>
<td>a: Unused --&gt; +a: Unused</td>
</tr>
</tbody>
</table>

Complexity

The overall complexity of the Plus is defined by the complexity of its subject, a, multiplied by the number of repetitions. The complexity of the Plus itself is O(N), where N is the number successful repetitions.

Example

```
Note

The test harness for the example(s) below is presented in the Basics Examples section.
```

Some using declarations:

```cpp
using boost::spirit::ascii::alpha;
using boost::spirit::qi::lexeme;
```

Parse one or more strings containing one or more alphabetic characters and put them in a vector:

```cpp
std::vector<std::string> attr;
test_phrase_parser_attr("yaba daba doo", +lexeme[+alpha], attr);
```

Sequence Parser (a >> b)

Description

The sequence operator, a >> b, parses two or more operands (a, b, ... etc.), in sequence:
Header

// forwards to <boost/spirit/home/qi/operator/sequence.hpp>
#include <boost/spirit/include/qi_sequence.hpp>

Also, see Include Structure.

Model of

NaryParser

Notation

a, b  Parse

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in NaryParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &gt;&gt; b</td>
<td>Match a followed by b.</td>
</tr>
</tbody>
</table>

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &gt;&gt; b</td>
<td>a: A, b: B ==&gt; (a &gt;&gt; b): tuple&lt;A, B&gt;</td>
</tr>
<tr>
<td></td>
<td>a: A, b: Unused ==&gt; (a &gt;&gt; b): A</td>
</tr>
<tr>
<td></td>
<td>a: Unused, b: B ==&gt; (a &gt;&gt; b): B</td>
</tr>
<tr>
<td></td>
<td>a: Unused, b: Unused ==&gt; (a &gt;&gt; b): Unused</td>
</tr>
<tr>
<td></td>
<td>a: A, b: A ==&gt; (a &gt;&gt; b): vector&lt;A&gt;</td>
</tr>
<tr>
<td></td>
<td>a: vector&lt;A&gt;, b: A ==&gt; (a &gt;&gt; b): vector&lt;A&gt;</td>
</tr>
<tr>
<td></td>
<td>a: A, b: vector&lt;A&gt; ==&gt; (a &gt;&gt; b): vector&lt;A&gt;</td>
</tr>
<tr>
<td></td>
<td>a: vector&lt;A&gt;, b: vector&lt;A&gt; ==&gt; (a &gt;&gt; b): vector&lt;A&gt;</td>
</tr>
</tbody>
</table>

Complexity

The overall complexity of the sequence parser is defined by the sum of the complexities of its elements. The complexity of the sequence itself is O(N), where N is the number of elements in the sequence.

Example

Some using declarations:
using boost::spirit::ascii::char_;  
using boost::spirit::qi::_1;  
using boost::spirit::qi::_2;  
namespace bf = boost::fusion;

Note
The test harness for the example(s) below is presented in the Basics Examples section.

Simple usage:

```cpp
test_parser("xy", char_ >> char_);
```

Extracting the attribute tuple (using Boost.Fusion):

```cpp
bf::vector<char, char> attr;  
test_parser_attr("xy", char_ >> char_, attr);  
std::cout << bf::at_c<0>(attr) << ',' << bf::at_c<1>(attr) << std::endl;
```

Extracting the attribute vector (using STL):

```cpp
std::vector<char> vec;  
test_parser_attr("xy", char_ >> char_, vec);  
std::cout << vec[0] << ',' << vec[1] << std::endl;
```

Extracting the attributes using Semantic Actions (using Phoenix):

```cpp
test_parser("xy", (char_ >> char_)[std::cout << _1 << ',' << _2 << std::endl]);
```

Sequential Or Parser \(a || b\)

Description
The sequential-or operator, \(a || b\), matches \(a\) or \(b\) or \(a\) followed by \(b\). That is, if both \(a\) and \(b\) match, it must be in sequence; this is equivalent to \(a >> -b | b\):

\[
\begin{align*}
a & || b & || \ldots
\end{align*}
\]

Header

```
// forwards to <boost/spirit/home/qi/operator/sequential_or.hpp>  
#include <boost/spirit/include/qi_sequential_or.hpp>
```

Also, see Include Structure.

Model of

NaryParser

Notation

\[ a, b \rightarrow \text{Parser} \]
### Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in `NaryParser`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

### Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
</table>
| a || b      | a: A, b: B --> (a || b): tuple<optional<A>, optional<B>>  
|            | a: A, b: Unused --> (a || b): optional<A>  
|            | a: Unused, b: B --> (a || b): optional<B>  
|            | a: Unused, b: Unused --> (a || b): Unused  
|            | a: A, b: A --> (a || b): vector<optional<A>> |

### Note

The sequential-or parser behaves attribute-wise very similar to the plain sequence parser (a >> b) in the sense that it exposes the attributes of its elements separately. For instance, if you attach a semantic action to the whole sequential-or:

```cpp
(int_ || int_)[print_pair(_1, _2)]
```

the function object `print_pair` would be invoked with the attribute of the first `int_` (boost::optional<int>) as its first parameter and the attribute of the second `int_` (boost::optional<int> as well) as its second parameter.

### Complexity

The overall complexity of the sequential-or parser is defined by the sum of the complexities of its elements. The complexity of the sequential-or itself is O(N), where N is the number of elements in the sequence.

### Example

#### Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::qi::int_;  
```

Correctly parsing a number with optional fractional digits:
A naive but incorrect solution would try to do this using optionals (e.g.):

```
int_ >> -(.' >> int_)  // will not match "123"
-int_ >> (.' >> int_)   // will not match "123"
-int_ >> -(.' >> int_)  // will match empty strings! Ooops.
```

### Stream Parsers

This module includes the description of the different variants of the `stream` parser. It can be used to utilize existing streaming operators (`operator>>(std::istream&, ...)`) for input parsing.

#### Header

```
// forwards to <boost/spirit/home/qi/stream.hpp>
#include <boost/spirit/include/qi_stream.hpp>
```

Also see Include Structure.

**Stream Parsers (stream, wstream, etc.)**

**Description**

The `stream_parser` is a primitive which allows to use pre-existing standard streaming operators for input parsing integrated with Spirit.Qi. It provides a wrapper parser dispatching the underlying input stream to the stream operator of the corresponding attribute type to be parsed. Any value `a` to be parsed using the `stream_parser` will result in invoking the standard streaming operator for its type `A`, for instance:

```
std::istream& operator>> (std::istream&, A&);
```

#### Header

```
// forwards to <boost/spirit/home/qi/stream.hpp>
#include <boost/spirit/include/qi_stream.hpp>
```

Also see Include Structure.

**Namespace**

**Name**

```
boost::spirit::stream // alias: boost::spirit::qi::stream
boost::spirit::wstream // alias: boost::spirit::qi::wstream
```
Synopsis

```cpp
template<typename Char, typename Attrib>
struct stream_parser;
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char</td>
<td>The character type to use to generate the input. This type will be used while assigning the generated characters to the underlying input iterator.</td>
<td>char</td>
</tr>
<tr>
<td>Attrib</td>
<td>The type of the attribute the stream_parser is expected to parse its input into.</td>
<td>spirit::basic_hold_any&lt;Char&gt;</td>
</tr>
</tbody>
</table>

Model of

**PrimitiveParser**

Notation

- A variable instance of any type with a defined matching streaming `operator>>()` or a **Lazy Argument** that evaluates to any type with a defined matching streaming `operator>>()`.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in **PrimitiveParser**.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>Call the streaming <code>operator&gt;&gt;(</code> for the type of the mandatory attribute. The input recognized by this operator will be the result of the stream parser. This parser never fails (unless the underlying input stream reports an error). The character type of the I/O istream is assumed to be char.</td>
</tr>
<tr>
<td>wstream</td>
<td>Call the streaming <code>operator&gt;&gt;(</code> for the type of the mandatory attribute. The input recognized by this operator will be the result of the wstream parser. This parser never fails (unless the underlying input stream reports an error). The character type of the I/O istream is assumed to be wchar_t.</td>
</tr>
</tbody>
</table>

All parsers listed in the table above are predefined specializations of the `stream_parser<Char>` basic stream parser type described below. It is possible to directly use this type to create stream parsers using an arbitrary underlying character type.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stream_parser&lt;Char, Attrib&gt;()</code></td>
<td>Call the streaming <code>operator&gt;&gt;(</code> for the type of the optional attribute, Attrib. The input recognized by this operator will be the result of the <code>stream_parser&lt;Char&gt;</code> parser. This parser never fails (unless the underlying input stream reports an error). The character type of the I/O istream is assumed to be Char</td>
</tr>
</tbody>
</table>
Additional Requirements

All of the stream parsers listed above require the type of the value to parse (the associated attribute) to implement a streaming operator conforming to the usual I/O streams conventions (where `attribute_type` is the type of the value to recognize while parse):

```cpp
template <typename Istream>
Istream& operator>>(Istream& os, attribute_type& attr)
{
    // type specific input parsing
    return os;
}
```

This operator will be called by the stream parsers to gather the input for the attribute of type `attribute_type`.

**Note**

If the stream parser is invoked inside a `match` (or `phrase_match`) stream manipulator the `Istream` passed to the `operator>>(Istream& os, attribute_type& attr)` will have registered (imbued) the same standard locale instance as the stream the `match` (or `phrase_match`) manipulator has been used with. This ensures all facets registered (imbued) with the original I/O stream object are used during input parsing.

Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stream</code></td>
<td><code>spirit::hold_any</code></td>
</tr>
<tr>
<td><code>wstream</code></td>
<td><code>spirit::whold_any</code></td>
</tr>
<tr>
<td><code>stream_parser&lt;Char, Attrib&gt;()</code></td>
<td><code>Attrib</code></td>
</tr>
</tbody>
</table>

**Important**

The attribute type `spirit::hold_any` exposed by some of the stream parsers is semantically and syntactically equivalent to the type implemented by `Boost.Any`. It has been added to `Spirit` as it has better performance and a smaller footprint than `Boost.Any`.

Complexity

O(N), where N is the number of characters consumed by the stream parser

Example

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

A class definition used in the examples:
// a simple complex number representation z = a + bi
struct complex
{
    complex (double a = 0.0, double b = 0.0)
    : a(a), b(b)
    {}

    double a;
    double b;
};

// define streaming operator for the type complex
std::istream&
operator>> (std::istream& is, complex& z)
{
    char lbrace = '\0', comma = '\0', rbrace = '\0';
    is >> lbrace >> z.a >> comma >> z.b >> rbrace;
    if (lbrace != '{' || comma != ',' || rbrace != '}')
        is.setstate(std::ios_base::failbit);
    return is;
}

Using declarations and variables:

using boost::spirit::qi::stream;
using boost::spirit::qi::stream_parser;

Parse a simple string using the operator>>(istream&, std::string&);

std::string str;
test_parser_attr("abc", stream, str);
std::cout << str << std::endl; // prints: abc

Parse our complex type using the operator>>(istream&, complex&);

complex c;
test_parser_attr("(1.0,2.5)", stream_parser<char, complex>()(), c);
std::cout << c.a << "," << c.b << std::endl; // prints: 1.0,2.5

String Parsers

This module includes parsers for strings. Currently, this module includes the literal and string parsers and the symbol table.

Module Header

// forwards to <boost/spirit/home/qi/string.hpp>
#include <boost/spirit/include/qi_string.hpp>

Also, see Include Structure.
String Parsers (string, lit)

Description

The string parser matches a string of characters. The string parser is an implicit lexeme: the skip parser is not applied in between characters of the string. The string parser has an associated Character Encoding Namespace. This is needed when doing basic operations such as inhibiting case sensitivity. Examples:

```cpp
string("Hello")
string(L"Hello")
string(s)  // s is a std::string
```

lit, like string, also matches a string of characters. The main difference is that lit does not synthesize an attribute. A plain string like "hello" or a std::basic_string is equivalent to a lit. Examples:

```cpp
"Hello"
lit("Hello")
lit(L"Hello")
lit(s)  // s is a std::string
```

Header

```cpp
// forwards to <boost/spirit/home/qi/string/lit.hpp>
#include <boost/spirit/include/qi_lit.hpp>
```

Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::lit</td>
</tr>
<tr>
<td>alias: boost::spirit::qi::lit</td>
</tr>
<tr>
<td>ns::string</td>
</tr>
</tbody>
</table>

In the table above, ns represents a Character Encoding Namespace.

Model of

PrimitiveParser

Notation

<table>
<thead>
<tr>
<th>s</th>
<th>A String or a Lazy Argument that evaluates to a String.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns</td>
<td>A Character Encoding Namespace.</td>
</tr>
</tbody>
</table>

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveParser.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>Create string parser from a string, s.</td>
</tr>
<tr>
<td>lit(s)</td>
<td>Create a string parser from a string, s.</td>
</tr>
<tr>
<td>ns::string(s)</td>
<td>Create a string parser with ns encoding from a string, s.</td>
</tr>
</tbody>
</table>
**Attributes**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>unused</td>
</tr>
<tr>
<td>lit(s)</td>
<td>unused</td>
</tr>
<tr>
<td>ns::string(s)</td>
<td>std::basic_string&lt;T&gt; where T is the underlying character type of s.</td>
</tr>
</tbody>
</table>

**Complexity**

O(N)

where N is the number of characters in the string to be parsed.

**Example**

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::qi::lit;
using boost::spirit::ascii::string;
```

Basic literals:

```cpp
test_parser("boost", "boost"); // plain literal
test_parser("boost", lit("boost")); // explicit literal
test_parser("boost", string("boost"); // ascii::string
```

From a std::string

```cpp
std::string s("boost");
test_parser("boost", s); // direct
test_parser("boost", lit(s)); // explicit
test_parser("boost", string(s)); // ascii::string
```

Lazy strings using Phoenix

```cpp
namespace phx = boost::phoenix;
test_parser("boost", phx::val("boost"); // direct
test_parser("boost", lit(phx::val("boost"))); // explicit
test_parser("boost", string(phx::val("boost"))); // ascii::string
```

**Symbols Parser (symbols)**

**Description**

The class symbols implements a symbol table: an associative container (or map) of key-value pairs where the keys are strings. The symbols class can work efficiently with 8, 16, 32 and even 64 bit characters.
Traditionally, symbol table management is maintained separately outside the grammar through semantic actions. Contrary to standard practice, the Spirit symbol table class `symbols` is-a parser, an instance of which may be used anywhere in the grammar specification. It is an example of a dynamic parser. A dynamic parser is characterized by its ability to modify its behavior at run time. Initially, an empty symbols object matches nothing. At any time, symbols may be added, thus, dynamically altering its behavior.

**Header**

```cpp
// forwards to <boost/spirit/home/qi/string/symbols.hpp>
#include <boost/spirit/include/qi_symbols.hpp>
```

Also, see Include Structure.

**Namespace**

```cpp
namespace boost::spirit::qi
{
    namespace symbols
    {
        // ...
    }
    namespace tst
    {
        // ...
    }
    namespace tst_map
    {
        // ...
    }
}
```

**Synopsis**

```cpp
template <typename Char, typename T, typename Lookup>
struct symbols;
```

**Template parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char</td>
<td>The character type of the symbol strings.</td>
<td>char</td>
</tr>
<tr>
<td>T</td>
<td>The data type associated with each symbol.</td>
<td>unused_type</td>
</tr>
<tr>
<td>Lookup</td>
<td>The symbol search implementation</td>
<td>tst&lt;Char, T&gt;</td>
</tr>
</tbody>
</table>

**Model of**

PrimitiveParser

**Notation**

- Sym  
  A symbols type.
- Char  
  A character type.
- T  
  A data type.
- sym, sym2  
  symbols objects.
- sseq  
  An STL container of strings.
- dseq  
  An STL container of data with value_type T.
- s1...sN  
  A String.
d1...dN    Objects of type \( T \).

\( f \)    A callable function or function object.

f, l    ForwardIterator first/last pair.

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in `PrimitiveParser`. 
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sym()</td>
<td>Construct an empty symbols named &quot;symbols&quot;.</td>
</tr>
<tr>
<td>Sym(name)</td>
<td>Construct an empty symbols named name.</td>
</tr>
<tr>
<td>Sym(sym2)</td>
<td>Copy construct a symbols from sym2 (Another symbols object).</td>
</tr>
<tr>
<td>Sym(sseq)</td>
<td>Construct symbols from sseq (an STL container of strings) named &quot;symbols&quot;.</td>
</tr>
<tr>
<td>Sym(sseq, name)</td>
<td>Construct symbols from sseq (an STL container of strings) named name.</td>
</tr>
<tr>
<td>Sym(sseq, dseq)</td>
<td>Construct symbols from sseq and dseq (An STL container of strings and an STL container of data with value_type T) which is named &quot;symbols&quot;.</td>
</tr>
<tr>
<td>Sym(sseq, dseq, name)</td>
<td>Construct symbols from sseq and dseq (An STL container of strings and an STL container of data with value_type T) which is named name.</td>
</tr>
<tr>
<td>sym = sym2</td>
<td>Assign sym2 to sym.</td>
</tr>
<tr>
<td>sym = s1, s2, ..., sN</td>
<td>Assign one or more symbols (s1...sN) to sym.</td>
</tr>
<tr>
<td>sym += s1, s2, ..., sN</td>
<td>Add one or more symbols (s1...sN) to sym.</td>
</tr>
<tr>
<td>sym.add(s1) (s2)...(sN)</td>
<td>Add one or more symbols (s1...sN) to sym.</td>
</tr>
<tr>
<td>sym.add(s1, d1)(s2, d2)...(sN, dN)</td>
<td>Add one or more symbols (s1...sN) with associated data (d1...dN) to sym.</td>
</tr>
<tr>
<td>sym -= s1, s2, ..., sN</td>
<td>Remove one or more symbols (s1...sN) from sym.</td>
</tr>
<tr>
<td>sym.remove(s1)(s2)...(sN)</td>
<td>Remove one or more symbols (s1...sN) from sym.</td>
</tr>
<tr>
<td>sym.clear()</td>
<td>Erase all of the symbols in sym.</td>
</tr>
<tr>
<td>sym.at(s)</td>
<td>Return a reference to the object associated with symbol, s. If sym does not already contain such an object, at inserts the default object T().</td>
</tr>
<tr>
<td>sym.find(s)</td>
<td>Return a pointer to the object associated with symbol, s. If sym does not already contain such an object, find returns a null pointer.</td>
</tr>
<tr>
<td>sym.prefix_find(f, l)</td>
<td>Return a pointer to the object associated with longest symbol that matches the beginning of the range [f, l), and updates f to point to one past the end of that match. If no symbol matches, then return a null pointer, and f is unchanged.</td>
</tr>
<tr>
<td>sym.for_each(f)</td>
<td>For each symbol in sym, s, a std::basic_string&lt;Char&gt; with associated data, d, an object of type T, invoke f(s, d)</td>
</tr>
<tr>
<td>sym.name()</td>
<td>Retrieve the current name of the symbols object.</td>
</tr>
</tbody>
</table>
**Expression**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sym.name(name)</code></td>
<td>Set the current name of the symbols object to be <code>name</code>.</td>
</tr>
</tbody>
</table>

**Attributes**

The attribute of `symbol<Char, T>` is `T`.

**Complexity**

The default implementation uses a Ternary Search Tree (TST) with complexity:

\[ O(\log n+k) \]

Where `k` is the length of the string to be searched in a TST with `n` strings.

TSTs are faster than hashing for many typical search problems especially when the search interface is iterator based. TSTs are many times faster than hash tables for unsuccessful searches since mismatches are discovered earlier after examining only a few characters. Hash tables always examine an entire key when searching.

An alternative implementation uses a hybrid hash-map front end (for the first character) plus a TST: `tst_map`. This gives us a complexity of

\[ O(1 + \log n+k-1) \]

This is found to be significantly faster than plain TST, albeit with a bit more memory usage requirements (each slot in the hash-map is a TST node). If you require a lot of symbols to be searched, use the `tst_map` implementation. This can be done by using `tst_map` as the third template parameter to the symbols class:

```cpp
symbols<Char, T, tst_map<Char, T> > sym;
```

**Example**

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::qi::symbols;
```

Symbols with data:
symbols<char, int> sym;

sym.add
    ("Apple", 1)
    ("Banana", 2)
    ("Orange", 3);

int i;
test_parser_attr("Banana", sym, i);
std::cout << i << std::endl;

When symbols is used for case-insensitive parsing (in a no_case directive), added symbol strings should be in lowercase. Symbol strings containing one or more uppercase characters will not match any input when symbols is used in a no_case directive.

symbols<char, int> sym;

sym.add
    ("apple", 1)  // symbol strings are added in lowercase...
    ("banana", 2)
    ("orange", 3);

int i;
// ...because sym is used for case-insensitive parsing
test_parser_attr("Apple", no_case[ sym ], i);
std::cout << i << std::endl;
test_parser_attr("ORANGE", no_case[ sym ], i);
std::cout << i << std::endl;

Qi Index

Symbols

_1 ... _N
 Phoenix, 80
_a ... _j
 Phoenix, 80
_pass
 Phoenix, 80
_r1 ... _rN
 Phoenix, 80
_val
 Phoenix, 80

A
alnum
 Character Classification Parsers (alnum, digit, etc.), 122
alpha
 Character Classification Parsers (alnum, digit, etc.), 122
alternative (a | b)
 Alternative Parser (a | b), 170
and-predicate (&a)
 And-Predicate Parser (& a), 172
as<T>
 Parser Directives Forcing Atomic Assignment (as<T>, as_string[], as_wstring[]), 139
as_string
 Parser Directives Forcing Atomic Assignment (as<T>, as_string[], as_wstring[]), 139
as_wstring
   Parser Directives Forcing Atomic Assignment (as<T>, as_string[], as_wstring[]), 139
attr_cast
   Attribute Transformation Pseudo Parser (attr_cast), 104
auto_
   Auto Parser, 99

B
big_dword
   Binary Big Endianness Parsers, 116
big_qword
   Binary Big Endianness Parsers, 116
big_word
   Binary Big Endianness Parsers, 116
bin
   Unsigned Integer Parsers (uint_, etc.), 148
blank
   Character Classification Parsers (alnum, digit, etc.), 122
bool_
   Boolean Parser (bool_), 166, 167
bool_parsers
   Boolean Parser (bool_), 166
bool_policies
   Boolean Parser (bool_), 166
byte_
   Binary Native Endianness Parsers, 112

C
char_
   Character Parser (char_, lit), 119
cntrl
   Character Classification Parsers (alnum, digit, etc.), 122
create_parser
   API for Automatic Parser Creation, 96
create_parser_exists
   API for Automatic Parser Creation, 96

D
difference (a - b)
   Difference Parser (a - b), 173
digit
   Character Classification Parsers (alnum, digit, etc.), 122
double_
   Real Number Parsers (float_, double_, etc.), 158
dword
   Binary Native Endianness Parsers, 112

E
eoi
   End of Input Parser (eoi), 107
eol
   End of Line Parser (eol), 106
eps
   Epsilon Parser (eps), 108
expectation (a > b)
   Expectation Parser (a > b), 175
F
  false_
    Boolean Parser ( bool_ ), 166
  float_
    Real Number Parsers ( float_ , double_ , etc.), 158

G
  grammar
    Parser Grammar, 145, 146
  graph
    Character Classification Parsers ( alnum , digit , etc.), 122

H
  hex
    Unsigned Integer Parsers ( uint_ , etc.), 148
  hold
    Parser Directive for Attribute Commit/Rollback ( hold [] ), 138

I
  inf
    Repetition Parser Directive ( repeat [] ), 132
  int_
    Signed Integer Parsers ( int_ , etc.), 153
  int_parser
    Signed Integer Parsers ( int_ , etc.), 153

K
  kleene (*a)
    Kleene Parser ( * a ), 176

L
  lazy
    Lazy Parser ( lazy ), 110, 111
  lexeme
    Parser Directive Inhibiting Skipping ( lexeme [] ), 125
    Parser Directive Inhibiting Skipping Without Pre-skip ( no_skip [] ), 126
    Parser Directive Re-Establishing Skipping ( skip [] ), 136
  list (a % b)
    List Parser ( a % b ), 178
  lit
    Boolean Parser ( bool_ ), 166
    Character Parser ( char_ , lit ), 119
    Real Number Parsers ( float_ , double_ , etc.), 158
    Signed Integer Parsers ( int_ , etc.), 153
    String Parsers ( string , lit ), 192
    Unsigned Integer Parsers ( uint_ , etc.), 148
  little_dword
    Binary Little Endianness Parser, 114
  little_qword
    Binary Little Endianness Parser, 114
  little_word
    Binary Little Endianness Parser, 114
  long_
    Signed Integer Parsers ( int_ , etc.), 153
  long_double
    Real Number Parsers ( float_ , double_ , etc.), 158
long_long
  Signed Integer Parsers ( int_ , etc.), 153
lower
  Character Classification Parsers ( alnum , digit , etc.), 122

M
match
  Stream Based Parser API, 93
matches
  Directive for Transduction Parsing ( raw [] ), 131
  Directive Testing if Parser Succeeded ( matches [] ), 135

N
not-predicate (!a)
  Not-Predicate Parser ( ! a ), 179
no_case
  Parser Directive Inhibiting Case Sensitivity ( no_case [] ), 128
no_skip
  Parser Directive Inhibiting Skipping ( lexeme [] ), 125
  Parser Directive Inhibiting Skipping Without Pre-skip ( no_skip [] ), 126
  Parser Directive Re-Establishing Skipping ( skip [] ), 136

O
oct
  Unsigned Integer Parsers ( uint_ , etc.), 148
omit
  Parser Directive Ignoring Attribute ( omit [] ), 129
optional (-a)
  Optional Parser ( - a ), 180

P
parse
  Iterator Based Parser API, 91
parser
  BinaryParser, 83, 84
  NaryParser, 84, 85
  Nonterminal, 85
  Parser, 81, 82
  Parser Concepts, 80
  PrimitiveParser, 82
  UnaryParser, 82, 83
permutation (a ^ b)
  Permutation Parser ( a ^ b ), 182
phrase_match
  Stream Based Parser API, 93
phrase_parse
  Iterator Based Parser API, 91
placeholders
  Phoenix, 80
plus (+a)
  Plus Parser ( + a ), 183
print
  Character Classification Parsers ( alnum , digit , etc.), 122
punct
  Character Classification Parsers ( alnum , digit , etc.), 122
q
qword
  Binary Native Endianness Parsers, 112

R
raw
  Directive for Transduction Parsing ( raw [] ), 131
real_parser
  Real Number Parsers ( float_ , double_ , etc.), 158
real_policies
  Predefined Policies, 158
  Real Number Parsers ( float_ , double_ , etc.), 158
repeat
  Repetition Parser Directive ( repeat [] ), 132
rule
  Parser Rule, 142, 143

S
Semantic Action
  Semantic Actions with Parsers, 98
sequence (a >> b)
  Sequence Parser ( a >> b ), 184
sequential-or (a || b)
  Sequential Or Parser ( a || b ), 186
short_
  Signed Integer Parsers ( int_ , etc.), 153
skip
  Parser Directive Inhibiting Skipping ( lexeme [] ), 125
  Parser Directive Inhibiting Skipping Without Pre-skip ( no_skip [] ), 126
  Parser Directive Re-Establishing Skipping ( skip [] ), 136
skip_flag::dont_postskip
  Iterator Based Parser API, 91
  Stream Based Parser API, 93
skip_flag::postskip
  Iterator Based Parser API, 91
  Stream Based Parser API, 93
space
  Character Classification Parsers ( alnum , digit , etc.), 122
string
  String Parsers ( string , lit ), 192
  Symbols Parser ( symbols ), 193, 194
symbols
  Symbols Parser ( symbols ), 193, 194

T
true_
  Boolean Parser ( bool_ ), 166

U
uint_
  Unsigned Integer Parsers ( uint_ , etc.), 148
uint_parser
  Real Number Parsers ( float_ , double_ , etc.), 158
  Signed Integer Parsers ( int_ , etc.), 153
  Unsigned Integer Parsers ( uint_ , etc.), 148
ulong_
Unsigned Integer Parsers (\texttt{uint\_}, etc.), 148
\texttt{ulong\_long}
Unsigned Integer Parsers (\texttt{uint\_}, etc.), 148
upper
  Character Classification Parsers (\texttt{alnum}, \texttt{digit}, etc.), 122
ureal\_policies
  Predefined Policies, 158
Real Number Parsers (\texttt{float\_}, \texttt{double\_}, etc.), 158
ushort_
  Unsigned Integer Parsers (\texttt{uint\_}, etc.), 148
\textbf{W}
word
  Binary Native Endianness Parsers, 112
\textbf{X}
xdigit
  Character Classification Parsers (\texttt{alnum}, \texttt{digit}, etc.), 122
Karma - Writing Generators

Tutorials

Quick Start

Spirit.Karma - what's that?

Throughout the description of Spirit.Karma we will try to align ourselves very much with the documentation for Spirit.Qi. The reasons are many fold:

- Spirit.Karma is the counterpart to Spirit.Qi. Some people say it's the Yin to Spirit.Qi's Yang. Spirit.Karma is generating byte sequences from internal data structures as Spirit.Qi is parsing byte sequences into those (very same) internal data structures.

- Both libraries have an almost identical structure, very similar semantics, and are both built using identical tools. Both libraries implement a language casting the specifics of their domain (parsing and generating) into a simple interface.

Why should you use a generator library for such a simple thing as output generation? Programmers have been using printf, std::stream formatting, or boost::format for quite some time. The answer is - yes, for simple output formatting tasks those familiar tools might be a quick solution. But experience shows: as soon as the formatting requirements are becoming more complex output generation is getting more and more challenging in terms of readability, maintainability, and flexibility of the code. Last, but not least, it turns out that code using Spirit.Karma runs much faster than equivalent code using either of the 'straight' methods mentioned above (see here for some numbers: Performance of Numeric Generators)

You might argue that more complex tasks require more complex tools. But this turns out not to be the case! The whole Spirit library is designed to be simple to use, while being scalable from trivial to very complicated applications.

In terms of development simplicity and ease in deployment, the same is true for Spirit.Karma as has been described elsewhere in this documentation for Spirit.Qi: the entire library consists of only header files, with no libraries to link against or build. Just put the spirit distribution in your include path, compile and run. Code size? Very tight, essentially comparable to hand written code.

The Spirit.Karma tutorials are built in a walk through style, starting with elementary things growing step by step in complexity. And again: keep in mind output generation is the exact opposite of parsing. Everything you already learnt about parsing using Spirit.Qi is applicable to generating formatted output using Spirit.Karma. All you have to do is to look at Spirit.Karma as being a mirror image of Spirit.Qi.

Warming up

Learning how to use Spirit.Karma is really simple. We will start from trivial examples, ramping up as we go.

Trivial Example #1 Generating a number

Let's create a generator that will output a floating-point number:

```
double_
```

Easy huh? The above code actually instantiates a Spirit floating point generator (a built-in generator). Spirit has many pre-defined generators and consistent naming conventions will help you finding your way through the maze. Especially important to note is that things related to identical entities (as in this case, floating point numbers) are named identically in Spirit.Karma and in Spirit.Qi. Actually, both libraries are using the very same variable instance to refer to a floating point generator or parser: double_.

Trivial Example #2 Generating two numbers

Now, let's create a generator that will output a line consisting of two floating-point numbers.
Here you see the familiar floating-point numeric generator `double_` used twice, once for each number. If you are used to see the `>>` operator for concatenating two parsers in `Spirit.Qi` you might wonder, what's that `<<` operator doing in there? We decided to distinguish generating and parsing of sequences the same way as the `std::stream` libraries do: we use operator `*>>` for input (parsing), and operator `*<<` for output (generating). Other than that there is no significant difference. The above program creates a generator from two simpler generators, gluing them together with the sequence operator. The result is a generator that is a composition of smaller generators. Whitespace between numbers can implicitly be inserted depending on how the generator is invoked (see below).

**Note**

When we combine generators, we end up with a "bigger" generator, but it's still a generator. Generators can get bigger and bigger, nesting more and more, but whenever you glue two generators together, you end up with one bigger generator. This is an important concept.

**Trivial Example #3 Generating one or more numbers**

Now, creating output for two numbers is not too interesting. Let's create a generator that will output zero or more floating-point numbers in a row.

```cpp
*double_
```

This is like a regular-expression Kleene Star. We moved the `*` to the front for the same reasons we did in `Spirit.Qi`: we must work with the syntax rules of C++. But if you know regular expressions (and for sure you remember those C++ syntax rules) it will start to look very familiar in a matter of a very short time.

Any expression that evaluates to a generator may be used with the Kleene Star. Keep in mind, though, that due to C++ operator precedence rules you may need to put the expression in parentheses for complex expressions. As above, whitespace can be inserted implicitly in between the generated numbers, if needed.

**Trivial Example #4 Generating a comma-delimited list of numbers**

We follow the lead of `Spirit.Qi`'s warming up section and will create a generator that produces a comma-delimited list of numbers.

```cpp
double_ << *(lit(',') << double_)
```

Notice `lit(',')`. It is a literal character generator that simply generates the comma `,`. In this case, the Kleene Star is modifying a more complex generator, namely, the one generated by the expression:

```cpp
(lit(',')) << double_
```

Note that this is a case where the parentheses are necessary. The Kleene Star encloses the complete expression above, repeating the whole pattern in the generated output zero or more times.

**Let's Generate!**

We're done with defining the generator. All that's left is to invoke the generator to do its work. For now, we will use the `generate_delimited` function. One overload of this function accepts four arguments:

1. An output iterator accepting the generated characters
2. The generator expression
3. Another generator called the delimiting generator
4. The data to format and output

While comparing this minimal example with an equivalent parser example we notice a significant difference. It is possible (and actually, it makes a lot of sense) to use a parser without creating any internal representation of the parsed input (i.e. without 'producing' any data from the parsed input). Using a parser in this mode checks the provided input against the given parser expression allowing to verify whether the input is parsable. For generators this mode doesn’t make any sense. What is output generation without generating any output? So we always will have to supply the data the output should be generated from. In our example we supply a list of double numbers as the last parameter to the function `generate_delimited` (see code below).

In this example, we wish to delimit the generated numbers by spaces. Another generator named `space` is included in Spirit’s repertoire of predefined generators. It is a very trivial generator that simply produces spaces. It is the equivalent to writing `lit(' ')`, or simply `' '`. It has been implemented for similarity with the corresponding predefined `space` parser. We will use `space` as our delimiter. The delimiter is the one responsible for inserting characters in between generator elements such as the `double_` and `lit`.

Ok, so now let’s generate (for the complete source code of this example please refer to `num_list1.cpp`).

```cpp
template <typename OutputIterator>
bool generate_numbers(OutputIterator& sink, std::list<double> const& v)
{
    using karma::double_;  
    using karma::generate_delimited;  
    using ascii::space;  

    bool r = generate_delimited(
        sink, // destination: output iterator
        double_ << *(',' << double_), // the generator
        space, // the delimiter-generator
        v // the data to output
    );
    return r;
}
```

**Note**

You might wonder how a `vector<double>`, which is actually a single data structure, can be used as an argument (we call it attribute) to a sequence of generators. This seems to be counter intuitive and doesn’t match with your experience of using `printf`, where each formatting placeholder has to be matched with a corresponding argument. Well, we will explain this behavior in more detail later in this tutorial. For now just consider this to be a special case, implemented on purpose to allow more flexible output formatting of STL containers: sequences accept a single container attribute if all elements of this sequence accept attributes compatible with the elements held by this container.

The `generate` function returns `true` or `false` depending on the result of the output generation. As outlined in different places of this documentation, a generator may fail for different reasons. One of the possible reasons is an error in the underlying output iterator (memory exhausted or disk full, etc.). Another reason might be that the data doesn’t match the requirements of a particular generator.
Note

char and wchar_t operands

The careful reader may notice that the generator expression has ',' instead of `lit(',)` as the previous examples did. This is ok due to C++ syntax rules of conversion. Spirit provides `<<` operators that are overloaded to accept a `char` or `wchar_t` argument on its left or right (but not both). An operator may be overloaded if at least one of its parameters is a user-defined type. In this case, the `double_` is the 2nd argument to `operator<<`, and so the proper overload of `<<` is used, converting ',' into a character literal generator.

The problem with omitting the `lit` should be obvious: `a' << 'b'` is not a spirit generator, it is a numeric expression, left-shifting the ASCII (or another encoding) value of 'a' by the ASCII value of 'b'. However, both `lit('a')` `<< 'b'` and `a' << lit('b')` are Spirit sequence generators for the letter 'a' followed by 'b'. You'll get used to it, sooner or later.

Note that we inlined the generator directly in the call to `generate_delimited`. Upon calling this function, the expression evaluates into a temporary, unnamed generator which is passed into the `generate_delimited` function, used, and then destroyed.

Here, we chose to make the generate function generic by making it a template, parameterized by the output iterator type. By doing so, it can put the generated data into any STL conforming output iterator.

Generator Semantic Actions

In the previous section we mentioned a very important difference between parsers and generators. While parsers may be used without 'producing' any data, generators always need data to generate the output from. We mentioned one way of passing data to the generator by supplying it as a parameter to one of the main API functions (for instance `generate()` or `generate_delimited()`). But sometimes this is not possible or not desirable.

Very much like for `Spirit.Qi` we have semantic actions in `Spirit.Karma` as well. Semantic actions may be attached to any point in the grammar specification. These actions are C++ functions or function objects that are called whenever a part of the generator is about to be invoked. Say you have a generator G, and a C++ function F, you can make the generator call F just before it gets invoked by attaching F:

```
G[F]
```

The expression above links F to the generator, G.

Semantic actions in `Spirit.Qi` are invoked after a parser successfully matches its input and the matched value is passed into the semantic action. In `Spirit.Karma` the opposite happens. Semantic actions are called before its associated generator is invoked. They may provide the data required by the generator.

The function/function object signature depends on the type of the generator to which it is attached. The generator `double_` expects the number to generate. Thus, if we were to attach a function F to `double_`, we need F to be declared as:

```
void F(double& n);
```

where the function is expected to initialize the parameter n with the value to generate.
Important

Generally, and more formally, the semantic action $F$ attached to a generator $G$ needs to take a reference to the generator's attribute type as its first parameter. For more information about generator attributes please see the section Generator Attributes.

In the example above the function $F$ takes a `double&` as its first parameter as the attribute of the `double_generator` happens to be a `double`.

There are actually 2 more arguments being passed (the generator context and a reference to a boolean 'pass' parameter). We don't need these, for now, but we'll see more on these other arguments later. *Spirit.Karma* allows us to bind a single argument function, like above. The other arguments are simply ignored.

To sum up, the possible signatures for semantic actions are:

```
void f(Attribs);
void f(Attribs, Contexts);
void f(Attribs, Contexts, bools);
```

Examples of Semantic Actions

In the following example we present various ways to attach semantic actions:

- Using a plain function pointer
- Using a simple function object
- Using *Boost.Bind* with a plain function
- Using *Boost.Bind* with a member function
- Using *Boost.Lambda*

Let's assume we have:
namespace client
{
    namespace karma = boost::spirit::karma;

    // A plain function
    void read_function(int& i)
    {
        i = 42;
    }

    // A member function
    struct reader
    {
        void print(int& i) const
        {
            i = 42;
        }
    };

    // A function object
    struct read_action
    {
        void operator()(int& i, unused_type, unused_type) const
        {
            i = 42;
        }
    };
}

Take note that with function objects, we need to have an operator() with 3 arguments. Since we don't care about the other two, we can use unused_type for these. We'll see more of unused_type elsewhere. Get used to it. unused_type is a Spirit supplied support class. Most of the time it stands for 'I don't care, just use the appropriate default'.

All following examples generate outputs of the form:

"{integer}"

An integer inside the curly braces.

The first example shows how to attach a plain function:

```
generate(outiter, '{' << int_['read_function'] << '}');
```

What's new? Well int_ is the sibling of double_. I'm sure you can guess what this generator does and what type of attribute it expects.

The next example shows how to attach a simple function object:

```
generate(outiter, '{' << int_[read_action()] << '}');
```

We can use Boost.Bind to 'bind' member functions:

```
reader r;
generate(outiter, '{' << int_[boost::bind(&reader::print, &r, _1)] << '}');
```

Likewise, we can also use Boost.Bind to 'bind' plain functions:
generate(outiter, '{' << int_[boost::bind(&read_function, _1)] << '}');

And last but not least, we can also use Boost.Lambda:

```cpp
std::stringstream strm("42");
generate(outiter, '{' << int_[strm >> lambda::_1] << '}');
```

There are more ways to bind semantic action functions, but the examples above are the most common. Attaching semantic actions is the first hurdle one has to tackle when getting started with generating with Spirit. If you didn't do so yet, it is probably a good idea to familiarize yourself with the tools behind it such as Boost.Bind and Boost.Lambda.

The examples above can be found here: actions.cpp

Phoenix

Phoenix, a companion library bundled with Spirit, is specifically suited for binding semantic actions. It is like Boost.Lambda on steroids, with special custom features that make it easy to integrate semantic actions with Spirit. If your requirements go beyond simple to moderate generation, I suggest you use this library. Examples presented henceforth shall be using the Phoenix library exclusively.

Important

There are different ways to write semantic actions for Spirit.Karma: using plain functions, Boost.Bind, Boost.Lambda, or Phoenix. The latter three allow you to use special placeholders to control parameter placement (_1, _2, etc.). Each of those libraries has it's own implementation of the placeholders, all in different namespaces. You have to make sure not to mix placeholders with a library they don't belong to and not to use different libraries while writing a semantic action.

Generally, for Boost.Bind, use ::_1, ::_2, etc. (yes, these placeholders are defined in the global namespace).

For Boost.Lambda use the placeholders defined in the namespace boost::lambda.

For semantic actions written using Phoenix use the placeholders defined in the namespace boost::spirit. Please note that all existing placeholders for your convenience are also available from the namespace boost::spirit::karma.

Complex - A first more complex generator

In this section we will develop a generator for complex numbers, allowing to represent a `std::complex` either as (real, imag) (where real and imag are the real and imaginary parts of the complex number) or as a simple real if the imaginary part happens to be equal to zero. This example will highlight the power of Spirit.Karma allowing to combine compile time definition of formatting rules with runtime based decisions which of the rules to apply. Also this time, we're using Boost.Phoenix to do the semantic actions.

Our goal is to allow for two different output formats to be applied depending on whether the imaginary part of the complex number is zero or not. Let's write both as a set of alternatives:

```
'(' << double_ << ", " << double_ << ')'
| double_
```

where the first alternative should be used for numbers having a non-zero imaginary part, while the second is for real numbers. Generally, alternatives are tried in the sequence of their definition as long until one of the expressions (as delimited by '|' ') succeeds. If no generator expression succeeds the whole alternative fails.

If we left this formatting grammar as is our generator would always choose the first alternative. We need to add some additional rules allowing to make the first alternative fail. So, if the first alternative fails the second one will be chosen instead. The decision about whether to choose the first alternative has to be made at runtime as only then we actually know the value of the imaginary part.
of the complex number. *Spirit.Karma* provides us with with a primitive generator `eps()`, which is usable as a semantic predicate. It has the property to 'succeed' generating only if its argument is true (while it never generates any output on its own).

```cpp
double imag = ...;     // imaginary part
eps(imag != 0) << '(' << double_<-_1 = c.real()>> "", " << double_<-_1 = c.imag()>> ')' | double_<-_1 = c.real())
```

If one of the generator elements of a sequence fails the whole sequence will fail. This is exactly what we need, forcing the second alternative to be chosen for complex numbers with imaginary parts equal to zero.

Now on to the full example, this time with the proper semantic actions (the complete cpp file for this example can be found here: `complex_number.cpp`).

We will use the `std::complex` type for this and all subsequent related examples. And here you can see the full code of the generator allowing to output a complex number either as a pair of numbers (if the imaginary part is non-zero) or as a single number (if the complex is a real number):

```cpp
template <typename OutputIterator>
bool generate_complex(OutputIterator sink, std::complex<double> const& c)
{
    using boost::spirit::karma::eps;
    using boost::spirit::karma::double_;  
    using boost::spirit::karma::_1;       
    using boost::spirit::karma::generate;

    return generate(sink,
        // Begin grammar
        (eps(c.imag() != 0) <<
         '(' << double_<-_1 = c.real()>> "", " << double_<-_1 = c.imag()>> ')' |
         double_<-_1 = c.real())
    )
        // End grammar
    );
}
```

The `double_` generators have this semantic action attached:

```cpp
_l = n
```

which passes `n` to the first element of the `s` generator's attached semantic action. Remember, semantic actions in *Spirit.Karma* are called before the corresponding generator is invoked and they are expected to provide the generator with the data to be used. The semantic action above assigns the value to be generated (`n`) to the generator (actually, the attribute of `double_<-_1`) is a Phoenix placeholder referring to the attribute of the semantic action's attached generator. If you need more information about semantic actions, you may want to read about them in this section: Semantic Actions.

These semantic actions are easy to understand but have the unexpected side effect of being slightly less efficient than it could be. In addition they tend to make the formatting grammar less readable. We will see in one of the next sections how it is possible to use other, built-in features of *Spirit.Karma* to get rid of the semantic actions altogether. When writing your grammars in Spirit you should always try to avoid semantic actions which is often possible. Semantic actions are really powerful tools but grammars tend to be more efficient and readable without them.

**Complex - Made easier**

In the previous section we showed how to format a complex number (i.e. a pair of doubles). In this section we will build on this example with the goal to avoid using semantic actions in the format specification. Let's have a look at the resulting code first, trying to understand it afterwards (the full source file for this example can be found here: `complex_number_easier.cpp`):
Let's cover some basic library features first.

**Making Numeric Generators Fail**

All Numeric Generators (such as `double_`, et.al.) take the value to emit from an attached attribute.

```cpp
double d = 1.5;
generate(out, double_, d); // will emit '1.5' (without the quotes)
```

Alternatively, they may be initialized from a literal value. For instance, to emit a constant 1.5 you may write:

```cpp
generate(out, double_(1.5)); // will emit '1.5' as well (without the quotes)
```

The difference to a simple 1.5 or `lit(1.5)` is that the `double_(1.5)` consumes an attribute if one is available. Additionally, it compares its immediate value to the value of the supplied attribute, and fails if those are not equal.

```cpp
double d = 1.5;
generate(out, double_(1.5), d); // will emit '1.5' as long as d == 1.5
```

This feature, namely to succeed generating only if the attribute matches the immediate value, enables numeric generators to be used to dynamically control the way output is generated.

**Note**

Quite a few generators will fail if their immediate value is not equal to the supplied attribute. Among those are all Character Generators and all String Generators. Generally, all generators having a sibling created by a variant of `lit()` belong into this category.

**Predicates - The Conditionals for Output Generators**

In addition to the `eps` generator mentioned earlier *Spirit.Karma* provides two special operators enabling dynamic flow control: the And predicate (unary `&`) and the Not predicate (unary `!`). The main property of both predicates is to discard all output emitted by the attached generator. This is equivalent to the behavior of predicates used for parsing. There the predicates do not consume any input allowing to look ahead in the input stream. In Karma, the and predicate succeeds as long as its associated generator succeeds, while the not predicate succeeds only if its associated generator fails.
The generator predicates in *Spirit.Karma* consume an attribute, if available. This makes them behave differently from predicates in *Spirit.Qi*, where they do not expose any attribute. This is because predicates allow to make decisions based on data available only at runtime. While in *Spirit.Qi* during parsing the decision is made based on looking ahead a few more input tokens, in *Spirit.Karma* the criteria has to be supplied by the user. The simplest way to do this is by providing an attribute.

As an example, the following generator succeeds generating

```cpp
double d = 1.0;
BOOST_ASSERT(generate(out, &double_(1.0), d)); // succeeds as d == 1.0
```

while this one will fail:

```cpp
double d = 1.0;
BOOST_ASSERT(!generate(out, !double_(1.0), d)); // fails as d == 1.0
```

Neither of these will emit any output. The predicates discard everything emitted by the generators to which they are applied.

### Ignoring Supplied Attributes

Sometimes it is desirable to 'skip' (i.e. ignore) a provided attribute. This happens for instance in alternative generators, where some of the alternatives need to extract only part of the overall attribute passed to the alternative generator. *Spirit.Karma* has a special pseudo generator for that: the directive `omit[]`. This directive consumes an attribute of the type defined by its embedded generator but it does not emit any output.

As we pass the imaginary part of the complex number as the attribute value for the `!double_(0.0)`, the overall first alternative will be chosen only if it is not equal to zero (the `!double_(0.0)` does not fail). That is exactly what we need!

Now, the second alternative has to emit the real part of the complex number only. In order to simplify the overall grammar we strive to unify the attribute types of all alternatives. As the attribute type exposed by the first alternative is `tuple<double, double, double>`, we need to skip the first and last element of the attribute (remember, we pass the real part as the second attribute element). We achieve this by using the `omit[]` directive:

```cpp
!double_(0.0) << '(' << double_ << "\", " << double_ << ')'
```
The overall attribute of this expression is `tuple<double, double, double>`, but the `omit[]` 'eats up' the first and the last element. The output emitted by this expression consist of a single generated double representing the second element of the tuple, i.e. the real part of our complex number.

---

### Important

Generally, it is preferable to use generator constructs not requiring semantic actions. The reason is that semantic actions often use constructs like: `double_*._1 = c.real()`. But this assignment is a real one! The data is in fact copied to the attribute value of the generator attached to the action. On the other hand, grammars without any semantic actions usually don't have to copy the attributes, making them more efficient.

---

### Complex - Fully Integrated

Until now, we have been working around the fact that `std::complex<>` is not a native Boost.Fusion sequence. We have not been able to use it with the same simplicity and natural grace of a `fusion::tuple<>` or a similar Boost.Fusion data structure. Fortunately, starting with Boost V1.43 it is possible to adapt any data structure (not only, as before, structures with publicly accessible members) as a Boost.Fusion sequence. All we have to do is to employ one of the new `BOOST_FUSION_ADAPT_CLASS` macros.

#### Adapting a Class As a Fusion Sequence

Let us start with the code again, following up with the explanations afterwards.

Wouldn't it be optimal if we could pass our instance of a `std::complex<>` directly to Karma's `generate()` function:

```cpp
template <typename OutputIterator>
bool generate_complex(OutputIterator sink, std::complex<double> const& c)
{
    using boost::spirit::karma::double_;
    using boost::spirit::karma::bool_;
    using boost::spirit::karma::true_;
    using boost::spirit::karma::omit;
    using boost::spirit::karma::generate;

    return generate(sink,
        // Begin grammar
        (true_ << '(' << double_ << ', ' << double_ << ')'
        | omit[bool_] << double_)
        ,
        // End grammar
        c // Data to output
    );
}
```

Indeed, this is possible! All we have to supply to make this work is a magic incantation (somewhere in the global namespace):
// We can leave off the setters as Karma does not need them.
BOOST_FUSION_ADAPT_ADT(
    std::complex<double>,
    (bool, bool, obj.imag() != 0, /**/)
    (double, double, obj.real(), /**/)
    (double, double, obj.imag(), /**/)
)

Most of the formatting grammar itself has not changed from the last section. We still utilize a very similar scheme. We have an alternative providing the formatting rules for our both use cases: one for the full complex format and one for complex numbers with a zero imaginary part. But instead of selecting the required alternative by comparing the imaginary part to zero in the grammar we assume to receive a boolean attribute carrying this information:

```cpp
&true_ << "(" << double_ << ", " << double_ << ")"
```

This reads as: 'if the first (boolean) element of the supplied fusion sequence is true, proceed as specified, else select the next alternative'. The next alternative now accounts for the boolean element as well, but is otherwise (almost) unchanged from the last section's example.

Now it should be clear why our adapt construct above exposes a three element Boost.Fusion sequence: a boolean and two double values (the real and the imaginary part of the complex number). We want it to match the requirements of our formatting grammar, which expects those exact values. The BOOST_FUSION_ADAPT_CLASS macro allows us to specify an arbitrary accessor construct, not necessarily limited to just calling a member function of the object instance (represented by obj in the context of this macro). This allows us to nicely encapsulate the decision logic into the class adaptation.

Here is the last new bit of information. If you look closely you realize the second alternative to be 'shorter' than the first one. It consumes only two elements of the supplied fusion sequence: it ignores the boolean and uses the real part of the complex number to generate its output. If there are more elements in our attribute than needed, we now can safely omit them from the grammar (which is a new 'feature' added to Spirit in V1.43 as well). Note, we could have written the alternative as

```cpp
&false_ << double_
```

but this would have been a bit less efficient as we needed to compare the boolean value again, while the final solution provided will just ignore it.

**Number List - Printing Numbers From a std::vector**

**Using the List Operator**

The C++ Standard library lacks an important feature, namely the support for any formatted output of containers. Sure, it's fairly easy to write a custom routine to output a specific container, but doing so over and over again is tedious at best. In this section we will demonstrate some more of the capabilities of Spirit.Karma for generating output from arbitrary STL containers. We will build on the example presented in an earlier section (see Warming Up).

The full source code of the example shown in this section can be found here: num_list2.cpp.

This time we take advantage of Karma's List (%) operator. The semantics of the list operator are fully equivalent to the semantics of the sequence we used before. The generator expression

```cpp
double_ << *(\',\' << double_)
```

is semantically equivalent to the generator expression
simplifying the overall code. The list operator's attribute is compatible with any STL container as well. For a change we use a `std::vector<double>` instead of the `std::list<double>` we used before. Additionally, the routine `generate_numbers` takes the container as a template parameter, so it will now work with any STL container holding `double` numbers.

```cpp
template <typename OutputIterator, typename Container>
bool generate_numbers(OutputIterator& sink, Container const& v)
{
    using boost::spirit::karma::double_;
    using boost::spirit::karma::generate_delimited;
    using boost::spirit::ascii::space;

    bool r = generate_delimited(
        sink,                      // destination: output iterator
        double_ % ',',           // the generator
        space,                   // the delimiter-generator
        v                         // the data to output
    );
    return r;
}
```

**Note**

Despite the container being a template parameter, the Spirit.Karma formatting expression (`double_ % ','`) does not depend on the actual type of the passed container. The only precondition to be met here is that the elements stored in the container have to be convertible to `double`.

**Generate Output from Arbitrary Data**

The output routine developed above is still not generically usable for all types of STL containers and for arbitrary elements stored in them. In order to be usable the items stored in the container still need to be convertible to a `double`. Fortunately Spirit.Karma is capable to output arbitrary data types while using the same format description expression. It implements the stream generators which are able to consume any attribute type as long as a matching standard streaming operator is defined. I.e. for any attribute type `Attrib` a function:

```cpp
std::ostream& operator<< (std::ostream&, Attrib const&);
```

needs to be available. The stream generator will use the standard streaming operator to generate the output.

The following example modifies the code shown above to utilize the stream operator, which makes it compatible with almost any data type. We implement a custom data type `complex` to demonstrate this. The example shows how it is possible to integrate this (or any other) custom data type into the Spirit.Karma generator framework.

This is the custom data structure together with the required standard streaming operator:
And this is the actual call to generate the output from a vector of those. This time we interleave the generated output with newline breaks (see `eol`), putting each complex number onto a separate line:

```cpp
template <typename OutputIterator, typename Container>
bool generate_numbers(OutputIterator& sink, Container const& v)
{
    using boost::spirit::karma::stream;
    using boost::spirit::karma::generate;
    using boost::spirit::karma::eol;

    bool r = generate(
        sink, // destination: output iterator
        stream % eol, // the generator
        v // the data to output
    );
    return r;
}
```

The code shown is fully generic and can be used with any STL container as long as the data items stored in that container implement the standard streaming operator.

The full source code of the example presented in this section can be found here: `num_list3.cpp`.

### Matrix of Numbers - Printing Numbers From a Matrix

In this section we will discuss the possibilities of `Spirit.Karma` when it comes to generating output from more complex - but still regular - data structures. For simplicity we will use a `std::vector<std::vector<int>>` as a poor man’s matrix representation. But even if the data structure seems to be very simple, the presented principles are applicable to more complex, or custom data structures as well. The full source code of the example discussed in this section can be found here: `num_matrix.cpp`.

### Quick Reference

This quick reference section is provided for convenience. You can use this section as sort of a "cheat-sheet" on the most commonly used Karma components. It is not intended to be complete, but should give you an easy way to recall a particular component without having to dig up on pages upon pages of reference documentation.

### Common Notation

#### Notation

\[ G \]  
Generator type
Generator objects

Attribute types of generators a, b, c, and d

The iterator type used for generation

An unused_type

The enclosing rule's Context type

An attribute value

An attribute type

A boolean expression

A type to be interpreted in boolean expressions

A (lazy generator) function with signature G(Unused, Context)

A (semantic action) function with signature void(Attrib&, Context, bool&). The third parameter is a boolean flag that can be set to false to force the generator to fail. Both Context and the boolean flag are optional.

An output iterator to receive the generated output

Character-class specific character type (See Character Class Types)

Character-class specific character (See Character Class Types)

Character-set specifier string (example: "a-z0-9")

Character-class specific string (See Character Class Types)

Attribute of str: std::basic_string<T> where T is the underlying character type of str

Numeric literal, any integer or real number type

Attribute of num: any integer or real number type

Used as a placeholder for a fusion sequence

Used as a placeholder for an STL container

Used as a placeholder for a boost::variant

Used as a placeholder for a boost::optional

Karma Generators

Character Generators

See here for more information about Character Generators.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>Unused</td>
<td>Generate ch</td>
</tr>
<tr>
<td>lit(ch)</td>
<td>Unused</td>
<td>Generate ch</td>
</tr>
<tr>
<td>char_</td>
<td>Ch</td>
<td>Generate character supplied as the attribute</td>
</tr>
<tr>
<td>char_(ch)</td>
<td>Ch</td>
<td>Generate ch, if an attribute is supplied it must match</td>
</tr>
<tr>
<td>char_(&quot;c&quot;)</td>
<td>Ch</td>
<td>Generate a single char string literal, c, if an attribute is supplied it must match</td>
</tr>
<tr>
<td>char_(ch, ch2)</td>
<td>Ch</td>
<td>Generate the character supplied as the attribute, if it belongs to the character range from ch to ch2</td>
</tr>
<tr>
<td>char_(charset)</td>
<td>Ch</td>
<td>Generate the character supplied as the attribute, if it belongs to the character set charset</td>
</tr>
<tr>
<td>alnum</td>
<td>Ch</td>
<td>Generate the character supplied as the attribute if it satisfies the concept of std::isalnum in the character set defined by NS</td>
</tr>
<tr>
<td>alpha</td>
<td>Ch</td>
<td>Generate the character supplied as the attribute if it satisfies the concept of std::isalpha in the character set defined by NS</td>
</tr>
<tr>
<td>blank</td>
<td>Ch</td>
<td>Generate the character supplied as the attribute if it satisfies the concept of std::isblank in the character set defined by NS</td>
</tr>
<tr>
<td>cntrl</td>
<td>Ch</td>
<td>Generate the character supplied as the attribute if it satisfies the concept of std::iscntrl in the character set defined by NS</td>
</tr>
<tr>
<td>digit</td>
<td>Ch</td>
<td>Generate the character supplied as the attribute if it satisfies the concept of std::isdigit in the character set defined by NS</td>
</tr>
<tr>
<td>graph</td>
<td>Ch</td>
<td>Generate the character supplied as the attribute if it satisfies the concept of std::isgraph in the character set defined by NS</td>
</tr>
<tr>
<td>print</td>
<td>Ch</td>
<td>Generate the character supplied as the attribute if it satisfies the concept of std::isprint in the character set defined by NS</td>
</tr>
<tr>
<td>Expression</td>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>punct</td>
<td>Ch</td>
<td>Generate the character supplied as the attribute if it satisfies the concept of ( \text{std::ispunct} ) in the character set defined by NS.</td>
</tr>
<tr>
<td>space</td>
<td>Ch</td>
<td>Generate the character supplied as the attribute if it satisfies the concept of ( \text{std::isspace} ), or a single space character in the character set defined by NS.</td>
</tr>
<tr>
<td>xdigit</td>
<td>Ch</td>
<td>Generate the character supplied as the attribute if it satisfies the concept of ( \text{std::isxdigit} ) in the character set defined by NS.</td>
</tr>
<tr>
<td>lower</td>
<td>Ch</td>
<td>Generate the character supplied as the attribute if it satisfies the concept of ( \text{std::islower} ) in the character set defined by NS.</td>
</tr>
<tr>
<td>upper</td>
<td>Ch</td>
<td>Generate the character supplied as the attribute if it satisfies the concept of ( \text{std::isupper} ) in the character set defined by NS.</td>
</tr>
</tbody>
</table>

**String Generators**

See here for more information about String Generators.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>Unused</td>
<td>Generate str</td>
</tr>
<tr>
<td>lit(str)</td>
<td>Unused</td>
<td>Generate str</td>
</tr>
<tr>
<td>string</td>
<td>Str</td>
<td>Generate string supplied as the attribute</td>
</tr>
<tr>
<td>string(str)</td>
<td>Str</td>
<td>Generate str, if an attribute is supplied it must match</td>
</tr>
<tr>
<td>symbols&lt;Attr, T&gt;</td>
<td>Attr</td>
<td>Declare a symbol table, sym.Attr is the The type of the original attribute to be used as the key into the symbol generator. T is the data type associated with each key.</td>
</tr>
<tr>
<td>sym.add</td>
<td>N/A</td>
<td>Add symbols into a symbol table, sym. val1 and val2 are optional data of type T, the data type associated with each key.</td>
</tr>
<tr>
<td>sym</td>
<td>T</td>
<td>Emit entries in the symbol table, sym. If attribute is found in the symbol table, the corresponding value is emitted. If sym does not store values, the original attribute is emitted.</td>
</tr>
</tbody>
</table>

**Real Number Generators**

See here for more information about Numeric Generators.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lit(num)</td>
<td>Unused</td>
<td>Generate num</td>
</tr>
<tr>
<td>float_</td>
<td>float</td>
<td>Generate a real number from a float</td>
</tr>
<tr>
<td>float_(num)</td>
<td>float</td>
<td>Generate num as a real number from a float, if an attribute is supplied it must match num</td>
</tr>
<tr>
<td>double_</td>
<td>double</td>
<td>Generate a real number from a double</td>
</tr>
<tr>
<td>double_(num)</td>
<td>double</td>
<td>Generate num as a real number from a double, if an attribute is supplied it must match num</td>
</tr>
<tr>
<td>long_double</td>
<td>long double</td>
<td>Generate a real number from a long double</td>
</tr>
<tr>
<td>long_double(num)</td>
<td>long double</td>
<td>Generate num as a real number from a long double, if an attribute is supplied it must match num</td>
</tr>
<tr>
<td>real_generator&lt;Num, Policies&gt;()</td>
<td>Num</td>
<td>Generate a real number Num using the supplied real number formatting policies</td>
</tr>
<tr>
<td>real_generator&lt;Num, Policies&gt;()(num)</td>
<td>Num</td>
<td>Generate real number num as a Num using the supplied real number formatting policies, if an attribute is supplied it must match</td>
</tr>
</tbody>
</table>
## Integer Generators

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lit(num)</code></td>
<td>Unused</td>
<td>Generate num</td>
</tr>
<tr>
<td><code>short_</code></td>
<td>short</td>
<td>Generate a short integer</td>
</tr>
<tr>
<td><code>short_(num)</code></td>
<td>short</td>
<td>Generate num as a short integer, if an attribute is supplied it must match</td>
</tr>
<tr>
<td><code>int_</code></td>
<td>int</td>
<td>Generate an int</td>
</tr>
<tr>
<td><code>int_(num)</code></td>
<td>int</td>
<td>Generate num as an int, if an attribute is supplied it must match</td>
</tr>
<tr>
<td><code>long_</code></td>
<td>long</td>
<td>Generate a long integer</td>
</tr>
<tr>
<td><code>long_(num)</code></td>
<td>long</td>
<td>Generate num as long integer, if an attribute is supplied it must match</td>
</tr>
<tr>
<td><code>long_long</code></td>
<td>long long</td>
<td>Generate a long long</td>
</tr>
<tr>
<td><code>long_long(num)</code></td>
<td>long long</td>
<td>Generate num as a long long, if an attribute is supplied it must match</td>
</tr>
<tr>
<td><code>int_generator&lt;</code></td>
<td>Num</td>
<td>Generate a Num</td>
</tr>
</tbody>
</table>
| Num, Radix, force_sign>
> ()                 | Num           | Generate a num as a Num, if an attribute is supplied it must match           |
## Unsigned Integer Generators

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lit(num)</code></td>
<td>Unused</td>
<td>Generate num</td>
</tr>
<tr>
<td><code>ushort_(num)</code></td>
<td>unsigned short</td>
<td>Generate an unsigned short integer</td>
</tr>
<tr>
<td><code>ushort_(num)</code></td>
<td>unsigned short</td>
<td>Generate num as an unsigned short integer, if an attribute is supplied it must match</td>
</tr>
<tr>
<td><code>uint_(num)</code></td>
<td>unsigned int</td>
<td>Generate an unsigned int</td>
</tr>
<tr>
<td><code>uint_(num)</code></td>
<td>unsigned int</td>
<td>Generate num as an unsigned int, if an attribute is supplied it must match</td>
</tr>
<tr>
<td><code>ulong_</code></td>
<td>unsigned long</td>
<td>Generate an unsigned long integer</td>
</tr>
<tr>
<td><code>ulong_(num)</code></td>
<td>unsigned long</td>
<td>Generate num as an unsigned long integer, if an attribute is supplied it must match</td>
</tr>
<tr>
<td><code>ulong_long</code></td>
<td>unsigned long long</td>
<td>Generate an unsigned long long</td>
</tr>
<tr>
<td><code>ulong_long_(num)</code></td>
<td>unsigned long long</td>
<td>Generate num as an unsigned long integer, if an attribute is supplied it must match</td>
</tr>
<tr>
<td><code>bin</code></td>
<td>unsigned int</td>
<td>Generate a binary integer from an unsigned int</td>
</tr>
<tr>
<td><code>oct</code></td>
<td>unsigned int</td>
<td>Generate an octal integer from an unsigned int</td>
</tr>
<tr>
<td><code>hex</code></td>
<td>unsigned int</td>
<td>Generate a hexadecimal integer from an unsigned int</td>
</tr>
<tr>
<td><code>uint_generator&lt; Num, Radix &gt;()</code></td>
<td>Num</td>
<td>Generate an unsigned Num</td>
</tr>
<tr>
<td><code>uint_generator&lt; Num, Radix &gt;()(num)</code></td>
<td>Num</td>
<td>Generate an unsigned num as a Num, if an attribute is supplied it must match</td>
</tr>
</tbody>
</table>
**Boolean Generators**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lit(num)</code></td>
<td>Unused</td>
<td>Generate num</td>
</tr>
<tr>
<td><code>bool_</code></td>
<td>bool</td>
<td>Generate a boolean</td>
</tr>
<tr>
<td><code>bool_(b)</code></td>
<td>bool</td>
<td>Generate b as a boolean, if an attribute is supplied it must match</td>
</tr>
<tr>
<td><code>bool_generator&lt; B, Policies &gt;()</code></td>
<td>B</td>
<td>Generate a boolean of type B</td>
</tr>
<tr>
<td><code>bool_generator&lt; B, Policies &gt;()(b)</code></td>
<td>B</td>
<td>Generate a boolean b as a B, if an attribute is supplied it must match</td>
</tr>
</tbody>
</table>

**Stream Generators**

See here for more information about Stream Generators.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stream</code></td>
<td><code>hold_any</code></td>
<td>Generate narrow character (char) based output using the matching streaming operator&lt;&lt;()</td>
</tr>
<tr>
<td><code>stream(s)</code></td>
<td><code>Unused</code></td>
<td>Generate narrow character (char) based output from the immediate arguments using the matching streaming operator&lt;&lt;()</td>
</tr>
<tr>
<td><code>wstream</code></td>
<td><code>whold_any</code></td>
<td>Generate wide character (wchar_t) based output using the matching streaming operator&lt;&lt;()</td>
</tr>
<tr>
<td><code>wstream(s)</code></td>
<td><code>Unused</code></td>
<td>Generate wide character (wchar_t) based output from the immediate arguments using the matching streaming operator&lt;&lt;()</td>
</tr>
<tr>
<td><code>stream_generator&lt;Char&gt;()</code></td>
<td><code>basic_holder_any&lt;Char&gt;</code></td>
<td>Generate output based on the given character type (Char) using the matching streaming operator&lt;&lt;()</td>
</tr>
<tr>
<td><code>stream_generator&lt;Char&gt;(s)</code></td>
<td><code>Unused</code></td>
<td>Generate output based on the given character type Char from the immediate arguments using the matching streaming operator&lt;&lt;()</td>
</tr>
</tbody>
</table>

**Binary Generators**

See here for more information about Binary Generators.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte_</td>
<td>8 bits native endian</td>
<td>Generate an 8 bit binary</td>
</tr>
<tr>
<td>word</td>
<td>16 bits native endian</td>
<td>Generate a 16 bit binary in native endian representation</td>
</tr>
<tr>
<td>big_word</td>
<td>16 bits big endian</td>
<td>Generate a 16 bit binary in big endian representation</td>
</tr>
<tr>
<td>little_word</td>
<td>16 bits little endian</td>
<td>Generate a 16 bit binary in little endian representation</td>
</tr>
<tr>
<td>dword</td>
<td>32 bits native endian</td>
<td>Generate a 32 bit binary in native endian representation</td>
</tr>
<tr>
<td>big_dword</td>
<td>32 bits big endian</td>
<td>Generate a 32 bit binary in big endian representation</td>
</tr>
<tr>
<td>little_dword</td>
<td>32 bits little endian</td>
<td>Generate a 32 bit binary in little endian representation</td>
</tr>
<tr>
<td>qword</td>
<td>64 bits native endian</td>
<td>Generate a 64 bit binary in native endian representation</td>
</tr>
<tr>
<td>big_qword</td>
<td>64 bits big endian</td>
<td>Generate a 64 bit binary in big endian representation</td>
</tr>
<tr>
<td>little_qword</td>
<td>64 bits little endian</td>
<td>Generate a 64 bit binary in little endian representation</td>
</tr>
<tr>
<td>pad(num)</td>
<td>Unused</td>
<td>Generate additional null bytes allowing to align generated output with memory addresses divisible by num.</td>
</tr>
</tbody>
</table>

**Auxiliary Generators**

See here for more information about Auxiliary Generators.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr_cast&lt;Exposed&gt;(a)</td>
<td>Exposed</td>
<td>Invoke a while supplying an attribute of type Exposed.</td>
</tr>
<tr>
<td>eol</td>
<td>Unused</td>
<td>Generate the end of line (\n)</td>
</tr>
<tr>
<td>eps</td>
<td>Unused</td>
<td>Generate an empty string</td>
</tr>
<tr>
<td>eps(b)</td>
<td>Unused</td>
<td>If b is true, generate an empty string</td>
</tr>
<tr>
<td>lazy(fg)</td>
<td>Attribute of G where G is the return type of fg</td>
<td>Invoke fg at generation time, returning a generator g which is then called to generate.</td>
</tr>
<tr>
<td>fg</td>
<td>see lazy(fg) above</td>
<td>Equivalent to lazy(fg)</td>
</tr>
</tbody>
</table>
**Auto Generators**

See here for more information about Auto Generators.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto_</td>
<td>hold_any</td>
<td>Generate output using a generator created from the supplied attribute type using the <code>create_generator</code> API function.</td>
</tr>
</tbody>
</table>

**Generator Operators**

See here for more information about Generator Operators.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!a</td>
<td>A</td>
<td>Not predicate. Ensure that a does not succeed generating, but don’t create any output</td>
</tr>
<tr>
<td>&amp;a</td>
<td>A</td>
<td>And predicate. Ensure that a does succeed generating, but don’t create any output</td>
</tr>
<tr>
<td>-a</td>
<td>optional&lt;A&gt;</td>
<td>Optional. Generate a zero or one time</td>
</tr>
<tr>
<td>*a</td>
<td>vector&lt;A&gt;</td>
<td>Kleene. Generate a zero or more times</td>
</tr>
<tr>
<td>+a</td>
<td>vector&lt;A&gt;</td>
<td>Plus. Generate a one or more times</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>variant&lt;A, B&gt;</td>
</tr>
<tr>
<td>a &lt;&lt; b</td>
<td>tuple&lt;A, B&gt;</td>
<td>Sequence. Generate a followed by b</td>
</tr>
<tr>
<td>a % b</td>
<td>vector&lt;A&gt;</td>
<td>List. Generate a delimited b one or more times</td>
</tr>
</tbody>
</table>

For more information about the attribute propagation rules implemented by the compound generators please see [Generator Compound Attribute Rules](#).

**Generator Directives**

See here for more information about Generator Directives.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lower[a]</code></td>
<td>A</td>
<td>Generate a as lower case</td>
</tr>
<tr>
<td><code>upper[a]</code></td>
<td>A</td>
<td>Generate a as upper case</td>
</tr>
<tr>
<td><code>left_align[a]</code></td>
<td>A</td>
<td>Generate a left aligned in column of width boosting_karma_default_field_length</td>
</tr>
<tr>
<td><code>left_align(num)[a]</code></td>
<td>A</td>
<td>Generate a left aligned in column of width num</td>
</tr>
<tr>
<td><code>left_align(g)[a]</code></td>
<td>A</td>
<td>Generate a left aligned in column of width boosting_karma_default_field_length while using g to generate the necessary padding</td>
</tr>
<tr>
<td><code>left_align(num, g)[a]</code></td>
<td>A</td>
<td>Generate a left aligned in column of width num while using g to generate the necessary padding</td>
</tr>
<tr>
<td><code>center[a]</code></td>
<td>A</td>
<td>Generate a centered in column of width boosting_karma_default_field_length</td>
</tr>
<tr>
<td><code>center(num)[a]</code></td>
<td>A</td>
<td>Generate a centered in column of width num</td>
</tr>
<tr>
<td><code>center(g)[a]</code></td>
<td>A</td>
<td>Generate a centered in column of width boosting_karma_default_field_length while using g to generate the necessary padding</td>
</tr>
<tr>
<td><code>center(num, g)[a]</code></td>
<td>A</td>
<td>Generate a centered in column of width num while using g to generate the necessary padding</td>
</tr>
<tr>
<td><code>right_align[a]</code></td>
<td>A</td>
<td>Generate a right aligned in column of width boosting_karma_default_field_length</td>
</tr>
<tr>
<td><code>right_align(num)[a]</code></td>
<td>A</td>
<td>Generate a right aligned in column of width num</td>
</tr>
<tr>
<td><code>right_align(g)[a]</code></td>
<td>A</td>
<td>Generate a right aligned in column of width boosting_karma_default_field_length while using g to generate the necessary padding</td>
</tr>
<tr>
<td><code>right_align(num, g)[a]</code></td>
<td>A</td>
<td>Generate a right aligned in column of width num while using g to generate the necessary padding</td>
</tr>
<tr>
<td><code>maxwidth[a]</code></td>
<td>A</td>
<td>Generate a truncated to column of width boosting_karma_default_field_max_width</td>
</tr>
</tbody>
</table>

229
<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>maxwidth(num)</code> [a]</td>
<td>A</td>
<td>Generate a truncated to column of width num</td>
</tr>
<tr>
<td><code>repeat</code> [a]</td>
<td>vector&lt;A&gt;</td>
<td>Repeat a zero or more times</td>
</tr>
<tr>
<td><code>repeat(num)</code> [a]</td>
<td>vector&lt;A&gt;</td>
<td>Repeat a num times</td>
</tr>
<tr>
<td><code>repeat(num1, num2)</code> [a]</td>
<td>vector&lt;A&gt;</td>
<td>Repeat a num1 to num2 times</td>
</tr>
<tr>
<td><code>repeat(num, inf)</code> [a]</td>
<td>vector&lt;A&gt;</td>
<td>Repeat a num or more times</td>
</tr>
<tr>
<td><code>delimit</code> [a]</td>
<td>A</td>
<td>Reestablish the delimiter that got inhibited by verbatim</td>
</tr>
<tr>
<td><code>delimit(d)</code> [a]</td>
<td>A</td>
<td>Use d as a delimiter for generating a</td>
</tr>
<tr>
<td><code>no_delimit</code> [a]</td>
<td>A</td>
<td>Disable delimited generation for a. No post-delimiting step performed.</td>
</tr>
<tr>
<td><code>as&lt;T&gt;[](a)</code> [a]</td>
<td>A</td>
<td>Force atomic output from arbitrary attribute types</td>
</tr>
<tr>
<td><code>as_string[](a)</code></td>
<td>A</td>
<td>Force atomic output from string attributes</td>
</tr>
<tr>
<td><code>as_wstring[](a)</code></td>
<td>A</td>
<td>Force atomic output from wide character string attributes</td>
</tr>
<tr>
<td><code>omit</code> [a]</td>
<td>A</td>
<td>Consume the attribute type of a without generating anything. The embedded generator will be always executed.</td>
</tr>
<tr>
<td><code>skip</code> [a]</td>
<td>A</td>
<td>Consume the attribute type of a without generating anything. The embedded generator will never be executed.</td>
</tr>
<tr>
<td><code>duplicate</code> [a]</td>
<td>A</td>
<td>The supplied attribute will be duplicated and passed unchanged to all embedded elements of a sequence.</td>
</tr>
<tr>
<td><code>buffer</code> [a]</td>
<td>A</td>
<td>Temporarily intercept the output generated by a, flushing it only after a succeeded.</td>
</tr>
<tr>
<td><code>columns</code> [a]</td>
<td>A</td>
<td>Generate a split into NUM_DEFAULT_COLUMNS number of columns using (\text{karma::eol}) as column delimiter</td>
</tr>
<tr>
<td><code>columns(num)</code> [a]</td>
<td>A</td>
<td>Generate a split into num number of columns using (\text{karma::eol}) as column delimiter</td>
</tr>
<tr>
<td>230</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>columns(g)[a]</code></td>
<td>A</td>
<td>Generate a split into <code>BOOST_KARMA_DEFAULT_COLUMNS</code> number of columns using <code>g</code> as column delimiter</td>
</tr>
<tr>
<td><code>columns(num, g)[a]</code></td>
<td>A</td>
<td>Generate a split into <code>num</code> number of columns using <code>g</code> as column delimiter</td>
</tr>
</tbody>
</table>

### Generator Semantic Actions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>g[fa]</code></td>
<td>Attribute of <code>g</code></td>
<td>Call semantic action <code>fa</code> before invoking <code>g</code></td>
</tr>
</tbody>
</table>

### Compound Attribute Rules

#### Notation

The notation we use is of the form:

```
  a: A, b: B, ... --> composite-expression: composite-attribute
```

- `a, b, etc.` are the operands. `A, B, etc.` are the operand's attribute types. `composite-expression` is the expression involving the operands and `composite-attribute` is the resulting attribute type of the composite expression.

For instance:

```
  a: A, b: B --> (a << b): tuple<A, B>
```

which reads as: given, `a` and `b` are generators, and `A` is the type of the attribute of `a`, and `B` is the type of the attribute of `b`, then the type of the attribute of `a << b` will be `tuple<A, B>`.

#### Important

In the attribute tables, we will use `vector<A>` and `tuple<A, B, ...>` as placeholders only. The notation of `vector<A>` stands for *any* `STL container` holding elements of type `A` and the notation `tuple<A, B, ...>` stands for *any* `Boost.Fusion sequence` holding `A, B, ...` etc. elements. The notation of `variant<A, B, ...>` stands for a `Boost.Variant` capable of holding `A, B, ...` etc. elements. Finally, `Unused` stands for `unused_type`.
### Compound Generator Attribute Types

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence ((a \ll b))</td>
<td>(a: A, b: B \rightarrow (a \ll b): \text{tuple}&lt;A, B&gt;)</td>
</tr>
<tr>
<td></td>
<td>(a: A, b: \text{Unused} \rightarrow (a \ll b): A)</td>
</tr>
<tr>
<td></td>
<td>(a: \text{Unused}, b: B \rightarrow (a \ll b): B)</td>
</tr>
<tr>
<td></td>
<td>(a: \text{Unused}, b: \text{Unused} \rightarrow (a \ll b): \text{Unused})</td>
</tr>
<tr>
<td></td>
<td>(a: A, b: A \rightarrow (a \ll b): \text{vector}&lt;A&gt;)</td>
</tr>
<tr>
<td></td>
<td>(a: \text{vector}&lt;A&gt;, b: A \rightarrow (a \ll b): \text{vector}&lt;A&gt;)</td>
</tr>
<tr>
<td></td>
<td>(a: A, b: \text{vector}&lt;A&gt; \rightarrow (a \ll b): \text{vector}&lt;A&gt;)</td>
</tr>
<tr>
<td></td>
<td>(a: \text{vector}&lt;A&gt;, b: \text{vector}&lt;A&gt; \rightarrow (a \ll b): \text{vector}&lt;A&gt;)</td>
</tr>
</tbody>
</table>

| Alternative \((a \mid b)\) | \(a: A, b: B \rightarrow (a \mid b): \text{variant}<A, B>\) |
| | \(a: A, b: \text{Unused} \rightarrow (a \mid b): A\) |
| | \(a: \text{Unused}, b: B \rightarrow (a \mid b): B\) |
| | \(a: \text{Unused}, b: \text{Unused} \rightarrow (a \mid b): \text{Unused}\) |
| | \(a: A, b: A \rightarrow (a \mid b): A\) |

| Kleene \(\ast a\) | \(a: A \rightarrow \ast a: \text{vector}<A>\) |
| | \(a: \text{Unused} \rightarrow \ast a: \text{Unused}\) |

| Plus (unary \(+\)) \(+a\) | \(a: A \rightarrow +a: \text{vector}<A>\) |
| | \(a: \text{Unused} \rightarrow +a: \text{Unused}\) |

| List \((\%\) \(a \% b\) | \(a: A, b: B \rightarrow (a \% b): \text{vector}<A>\) |
| | \(a: \text{Unused}, b: B \rightarrow (a \% b): \text{Unused}\) |

| Repetition (\(\text{repeat[]}\)) | \(a: A \rightarrow \text{repeat}(\ldots, \ldots)[a]: \text{vector}<A>\) |
| | \(a: \text{Unused} \rightarrow \text{repeat}(\ldots, \ldots)[a]: \text{Unused}\) |

| Optional (unary \(-\)) \(-a\) | \(a: A \rightarrow -a: \text{optional}<A>\) |
| | \(a: \text{Unused} \rightarrow -a: \text{Unused}\) |

| And predicate (unary \&\) \(\&a\) | \(a: A \rightarrow \&a: A\) |

| Not predicate (unary !) \(!a\) | \(a: A \rightarrow !a: A\) |

### Nonterminals

See here for more information about Nonterminals.

### Notation

\text{RT} \hspace{1cm} \text{Synthesized attribute. The rule or grammar's return type.}
Arg1, Arg2, ArgN  | Inherited attributes. Zero or more arguments.
L1, L2, LN       | Zero or more local variables.
r, r2             | Rules
q                 | A grammar
p                 | A generator expression
my_grammar        | A user defined grammar

**Terminology**

**Signature**  \( RT(Arg1, Arg2, \ldots, ArgN) \). The signature specifies the synthesized (return value) and inherited (arguments) attributes.

**Locals**      \( locals<L1, L2, \ldots, LN> \). The local variables.

**Delimiter**   The delimit-generator type

**Template Arguments**

**Iterator**    The iterator type you will use for parsing.

**A1, A2, A3**   Can be one of 1) Signature 2) Locals 3) Delimiter.
**Expression**  |  **Description**  
--- | ---  
`rule<OutputIterator, A1, A2, A3> r(name);` | Rule declaration. `OutputIterator` is required. `A1, A2, A3` are optional and can be specified in any order. `name` is an optional string that gives the rule its name, useful for debugging.  
`rule<OutputIterator, A1, A2, A3> r(r2);` | Copy construct rule `r` from rule `r2`.  
r = r2; | Assign rule `r2` to `r`. Boost::shared_ptr semantics.  
r.alias() | Return an alias of `r`. The alias is a generator that holds a reference to `r`. Reference semantics.  
r.copy() | Get a copy of `r`.  
r.name(name) | Set the name of a rule  
r.name() | Get the name of a rule  
debug(r) | Debug rule `r`  
r = g; | Rule definition  
r %= g; | Auto-rule definition. The attribute of `g` should be compatible with the synthesized attribute of `r`. When `g` is successful, its attribute is automatically propagated to `r`'s synthesized attribute.  

```cpp
template <typename OutputIterator>
struct my_grammar : grammar<OutputIterator, A1, A2, A3>  
{
    my_grammar() : my_gramm()
    {   
        my_grammar() : my_gramm
            {   
                // Rule definitions
                start = /* ... */;
            }
        rule<OutputIterator, A1, A2, A3> start;
    // more rule declarations...
    
};

my_grammar<OutputIterator> g  
instantiate a grammar

g.name(name)  
set the name of a grammar

g.name()  
get the name of a grammar
```

**Generator Semantic Actions**

Semantic Actions may be attached to any generator as follows:

```
g[f]
```

where `f` is a function with the signatures:
void f(Attrib&);
void f(Attrib&, Context&);
void f(Attrib&, Context&, bool&);

You can use Boost.Bind to bind member functions. For function objects, the allowed signatures are:

void operator()(Attrib&, unused_type, unused_type) const;
void operator()(Attrib&, Context&, unused_type) const;
void operator()(Attrib&, Context&, bool&) const;

The unused_type is used in the signatures above to signify 'don't care'.

For more information see Semantic Actions.

**Phoenix**

Boost.Phoenix makes it easier to attach semantic actions. You just inline your lambda expressions:

`g[phoenix-lambda-expression]`

**Spirit.Karma** provides some Boost.Phoenix placeholders to access important information from the Attrib and Context that are otherwise fiddly to extract.

**Spirit.Karma specific Phoenix placeholders**

- `_1, _2, ... , _N` Nth attribute of `g`
- `_val` The enclosing rule's synthesized attribute.
- `_r1, _r2, ... , _rN` The enclosing rule's Nth inherited attribute.
- `_a, _b, ... , _j` The enclosing rule's local variables (`_a` refers to the first).
- `_pass` Assign false to `_pass` to force a generator failure.

---

**Important**

All placeholders mentioned above are defined in the namespace `boost::spirit` and, for your convenience, are available in the namespace `boost::spirit::karma` as well.

For more information see Semantic Actions.

**Reference**

**Generator Concepts**

**Spirit.Karma** generators fall into a couple of generalized concepts. The `Generator` is the most fundamental concept. All **Spirit.Karma** generators are models of the `Generator` concept. `PrimitiveGenerator`, `UnaryGenerator`, `BinaryGenerator`, `NaryGenerator`, and `Nonterminal` are all refinements of the `Generator` concept.

The following sections provide details on these concepts.
Generator

Description

The Generator is the most fundamental concept. A Generator has a member function, generate, that accepts an OutputIterator and returns bool as its result. The iterator receives the data being generated. The Generator's generate member function returns true if the generator succeeds. Each Generator can represent a specific pattern or algorithm, or it can be a more complex generator formed as a composition of other Generators.

Notation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>A Generator.</td>
</tr>
<tr>
<td>G</td>
<td>A Generator type.</td>
</tr>
<tr>
<td>OutIter</td>
<td>An OutputIterator type.</td>
</tr>
<tr>
<td>sink</td>
<td>An OutputIterator instance.</td>
</tr>
<tr>
<td>Context</td>
<td>The generator's Context type.</td>
</tr>
<tr>
<td>context</td>
<td>The generator's Context, or unused.</td>
</tr>
<tr>
<td>delimit</td>
<td>A delimiter Generator, or unused.</td>
</tr>
<tr>
<td>attrib</td>
<td>A Compatible Attributes, or unused.</td>
</tr>
</tbody>
</table>

Valid Expressions

In the expressions below, the behavior of the generator, g, as well as how delimit and attrib are handled by g, are left unspecified in the base Generator concept. These are specified in subsequent, more refined concepts and by the actual models thereof.

For any Generator the following expressions must be valid:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>g.generate(sink, context, delimit, attrib)</td>
<td>Generate the output sequence by inserting the generated characters/tokens into sink. Use the delimit generator for delimiting. Return true if successful, otherwise return false.</td>
<td>bool</td>
</tr>
<tr>
<td>g.what(context)</td>
<td>Get information about a Generator.</td>
<td>info</td>
</tr>
</tbody>
</table>

Type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>G::template attribute&lt;Context&gt;::type</td>
<td>The Generator's attribute.</td>
</tr>
<tr>
<td>traits::is_generator&lt;G&gt;::type</td>
<td>Metaclass function that evaluates to mpl::true_ if a certain type, G is a Generator, mpl::false_ otherwise (See MPL Boolean Constant).</td>
</tr>
<tr>
<td>G::properties</td>
<td>An mpl::int_ (See MPL Integral Constant) holding a value from the karma::generator_properties enumeration. The default value is generator_properties::no_properties</td>
</tr>
</tbody>
</table>
Postcondition

Upon return from `g.generate` the following post conditions should hold:

- On successful generation, `sink` receives the generated characters/tokens sequence.
- No pre-delimits: `delimit` characters/tokens will not be emitted in front of any other output.
- The attribute `attrib` has not been modified.

Models

All generators in *Spirit.Karma* are models of the *Generator* concept.

**PrimitiveGenerator**

Description

*PrimitiveGenerator* is the most basic building block that the client uses to build more complex generators.

Refinement of

*Generator*

Post-delimit

Before exiting the `generate` member function, a `PrimitiveGenerator` is required to do a post-delimit. This will generate a single delimiting character/token sequence. Only `PrimitiveGenerator`'s are required to perform this post-delimit. This is typically carried out through a call to `karma::delimit_out`:

```
karma::delimit_out(sink, delimit);
```

Type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>traits::is_primitive_generator&lt;G&gt;::type</code></td>
<td>Metafunction that evaluates to <code>mpl::true_</code> if a certain type, <code>G</code>, is a <code>PrimitiveGenerator</code>, <code>mpl::false_</code> otherwise (See MPL Boolean Constant).</td>
</tr>
</tbody>
</table>

Models

The following generators conform to this model:

- `eol`,
- `eps`,
- Numeric generators,
- Character generators.

FIXME Add more links to *PrimitiveGenerator* models here.

**UnaryGenerator**

Description

*UnaryGenerator* is a composite generator that has a single subject. The `UnaryGenerator` may change the behavior of its subject following the Delegate Design Pattern.
Refinement of

Generator

Notation

g A UnaryGenerator.

G A UnaryGenerator type.

Valid Expressions

In addition to the requirements defined in Generator, for any UnaryGenerator the following must be met:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>g.subject</td>
<td>Subject generator.</td>
<td>Generator</td>
</tr>
</tbody>
</table>

Type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>G::subject_type</td>
<td>The subject generator type.</td>
</tr>
<tr>
<td>traits::is_unary_generator&lt;G&gt;::type</td>
<td>Metafunction that evaluates to mpl::true_ if a certain type,</td>
</tr>
<tr>
<td></td>
<td>G is a UnaryGenerator, mpl::false_ otherwise (See MPL Boolean Constant).</td>
</tr>
</tbody>
</table>

Invariants

For any UnaryGenerator, G, the following invariant always holds:

• traits::is_generator<G::subject_type>::type evaluates to mpl::true_

Models

The following generators conform to this model:

• Kleene Star (unary *) operator,
• Plus (unary +) operator,
• Optional (unary -) operator,
• And predicate (unary & ) and Not predicate (unary !) operators,
• left_align, center, and right_align directives,
• repeat directive,
• verbatim directive,
• delimit directive,
• lower and upper directives,
• maxWidth directive,
• buffer directive,
BinaryGenerator

Description

BinaryGenerator is a composite generator that has a two subjects, left and right. The BinaryGenerator allows its subjects to be treated in the same way as a single instance of a Generator following the Composite Design Pattern.

Refinement of

Generator

Notation

g  A BinaryGenerator.

G  A BinaryGenerator type.

Valid Expressions

In addition to the requirements defined in Generator, for any BinaryGenerator the following must be met:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>g.left</td>
<td>Left generator.</td>
<td>Generator</td>
</tr>
<tr>
<td>g.right</td>
<td>Right generator.</td>
<td>Generator</td>
</tr>
</tbody>
</table>

Type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>G::left_type</td>
<td>The left generator type.</td>
</tr>
<tr>
<td>G::right_type</td>
<td>The right generator type.</td>
</tr>
<tr>
<td>traits::is_binary_generator&lt;G&gt;::type</td>
<td>Metafunction that evaluates to mpl::true_ if a certain type, G is a BinaryGenerator, mpl::false_ otherwise (See MPL Boolean Constant).</td>
</tr>
</tbody>
</table>

Invariants

For any BinaryGenerator, G, the following invariants always hold:

• traits::is_generator<G::left_type>::type evaluates to mpl::true_

• traits::is_generator<G::right_type>::type evaluates to mpl::true_

Models

The following generators conform to this model:

• List (§).

FIXME Add more links to models of BinaryGenerator concept
NaryGenerator

Description

*NaryGenerator* is a composite generator that has one or more subjects. The NaryGenerator allows its subjects to be treated in the same way as a single instance of a *Generator* following the Composite Design Pattern.

Refinement of

Generator

Notation

g A NaryGenerator.

G A NaryGenerator type.

Valid Expressions

In addition to the requirements defined in *Generator*, for any NaryGenerator the following must be met:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>g.elements</td>
<td>The tuple of elements.</td>
<td>A Boost.Fusion Sequence of <em>Generator</em> types.</td>
</tr>
</tbody>
</table>

Type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>g.elements_type</td>
<td>Elements tuple type.</td>
</tr>
<tr>
<td>traits::is_nary_generator&lt;G&gt;::type</td>
<td>Metafunction that evaluates to mpl::true_ if a certain type, G is a NaryGenerator, mpl::false_ otherwise (See MPL Boolean Constant).</td>
</tr>
</tbody>
</table>

Invariants

For each element, E, in any NaryGenerator, G, the following invariant always holds:

- traits::is_generator<E>::type evaluates to mpl::true_

Models

The following generators conform to this model:

- Sequence (<<).
- Alternative (|).

FIXME Add more links to models of NaryGenerator concept

Nonterminal

Description

A Nonterminal is a symbol in a Parsing Expression Grammar production that represents a grammar fragment. Nonterminals may self reference to specify recursion. This is one of the most important concepts and the reason behind the word "recursive" in recursive descent generation.
Refinement of Generator

Signature

Rules can have both consumed and inherited attributes. The rule's *Signature* specifies both the consumed and inherited attributes. The specification uses the function declarator syntax:

\[
RT(A0, A1, A2, \ldots, AN)
\]

where \(RT\) is the rule's consumed attribute and \(A0 \ldots AN\) are the rule's inherited attributes.

Attributes

The rule models a C++ function. The rule's consumed attribute is analogous to the function return value as it is the type -exposed- by the rule. Its inherited attributes are analogous to function arguments. The inherited attributes (arguments) can be passed in just like any Lazy Argument, e.g.:

\[
r(expr) // Evaluate expr at parse time and pass the result to the Nonterminal r
\]

\_val

The `boost::spirit::karma::_val` placeholder can be used in Phoenix semantic actions anywhere in the Nonterminal's definition. This Phoenix placeholder refers to the Nonterminal's (consumed) attribute. The `_val` placeholder acts like an immutable reference to the Nonterminal's attribute.

Note

Starting with Spirit V2.5 (distributed with Boost V1.47) the placeholder `_val` can be used in semantic actions attached to top level generator components as well. See The Generator API for more information.

\_r1...r10

The `boost::spirit::r1...boost::spirit::r10` placeholders can be used in Phoenix semantic actions anywhere in the Nonterminal's definition. These Phoenix placeholders refer to the Nonterminal's inherited attributes.

Locals

Nonterminals can have local variables that will be created on the stack at runtime. A locals descriptor added to the Nonterminal declaration will give the Nonterminal local variables:

\[
\text{template <typename T0, typename T1, typename T2, \ldots, typename TN>}
\text{struct locals;}
\]

where \(T0 \ldots TN\) are the types of local variables accessible in your Phoenix semantic actions using the placeholders:

- `boost::spirit::_a`
- `boost::spirit::_b`
- `boost::spirit::_c`
- `boost::spirit::_d`
- `boost::spirit::_e`
- `boost::spirit::_f`
which correspond to the Nonterminal's local variables T0 ... T9.

**Notation**

\[ x \]
- A Nonterminal

\[ X \]
- A Nonterminal type

\[ arg1, arg2, ..., argN \]
- Lazy Arguments that evaluate to each of the Nonterminal's inherited attributes.

**Valid Expressions**

In addition to the requirements defined in *Generator*, for any Nonterminal the following must be met:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>In a generator expression, invoke Nonterminal ( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>( x(arg1, arg2, ..., argN) )</td>
<td>In a generator expression, invoke Nonterminal ( x ) passing in inherited attributes ( arg1...argN )</td>
<td>( x )</td>
</tr>
<tr>
<td>( x.name(name) )</td>
<td>Set the name of a Nonterminal</td>
<td>void</td>
</tr>
<tr>
<td>( x.name() )</td>
<td>Get the name of a Nonterminal</td>
<td>std::string</td>
</tr>
</tbody>
</table>

**Type Expressions**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X::sig_type )</td>
<td>The Signature of ( X ): An MPL Forward Sequence. The first element is the Nonterminal's consumed attribute type and the rest are the inherited attribute types.</td>
</tr>
<tr>
<td>( X::locals_type )</td>
<td>The local variables of ( X ): An MPL Forward Sequence.</td>
</tr>
</tbody>
</table>

**Models**

- rule
- grammar

**Generator Basics**

**Lazy Argument**

Some generators (e.g. primitives and non-terminals) may take in additional attributes. Such generators take the form:
\[ g(a_1, a_2, \ldots, a_N) \]

where \( g \) is a generator. Each of the arguments \((a_1 \ldots a_N)\) can either be an immediate value, or a function, \( f \), with signature:

\[ T \ f(\text{Unused}, \text{Context}) \]

where \( T \), the function's return value, is compatible with the argument type expected and \( \text{Context} \) is the generator's Context type (The first argument is \textit{unused} to make the \( \text{Context} \) the second argument. This is done for uniformity with \textit{Semantic Actions}).

**Character Encoding Namespace**

Some generators need to know which character set a \texttt{char} or \texttt{wchar_t} is operating on. For example, the \texttt{alnum} generator works differently with ISO8859.1 and ASCII encodings. Where necessary, Spirit encodes (tags) the generator with the character set.

We have a namespace for each character set Spirit will be supporting. That includes \texttt{ascii}, \texttt{iso8859_1}, \texttt{standard} and \texttt{standard_wide} (and in the future, \texttt{unicode}). In each of the character encoding namespaces, we place tagged versions of generators such as \texttt{alnum}, \texttt{space} etc.

Example:

```
using boost::spirit::ascii::space; // use the ASCII space generator
```

Namespaces:

- `boost::spirit::ascii`
- `boost::spirit::iso8859_1`
- `boost::spirit::standard`
- `boost::spirit::standard_wide`

For ease of use, the components in this namespaces are also brought into the karma sub-namespaces with the same names:

- `boost::spirit::karma::ascii`
- `boost::spirit::karma::iso8859_1`
- `boost::spirit::karma::standard`
- `boost::spirit::karma::standard_wide`

**Examples**

All sections in the reference present some real world examples. The examples use a common test harness to keep the example code as minimal and direct to the point as possible. The test harness is presented below.

Some includes:

```
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

The used output iterator:
typedef std::back_insert_iterator<std::string> output_iterator_type;

Our test functions:

This one tests the generators without attributes.

```cpp
template <typename G>
void test_generator(char const* expected, G const& g)
{
    std::string s;
    std::back_insert_iterator<std::string> out(s);
    if (boost::spirit::karma::generate(out, g) && s == expected)
        std::cout << "ok" << std::endl;
    else
        std::cout << "fail" << std::endl;
}
```

These test the generators with one or more user supplied attributes.

```cpp
template <typename G, typename T>
void test_generator_attr(char const* expected, G const& g, T const& attr)
{
    std::string s;
    std::back_insert_iterator<std::string> out(s);
    if (boost::spirit::karma::generate(out, g, attr) && s == expected)
        std::cout << "ok" << std::endl;
    else
        std::cout << "fail" << std::endl;
}
```

```cpp
template <typename G, typename T1, typename T2>
void test_generator_attr(char const* expected, G const& g, T1 const& attr1,
T2 const& attr2)
{
    std::string s;
    std::back_insert_iterator<std::string> out(s);
    if (boost::spirit::karma::generate(out, g, attr1, attr2) && s == expected)
        std::cout << "ok" << std::endl;
    else
        std::cout << "fail" << std::endl;
}
```

This tests the generators with one attribute and while using delimited output.

```cpp
template <typename G, typename Delimiter, typename T>
void test_generator_attr_delim(char const* expected, G const& g, Delimiter const& d, T const& attr)
{
    std::string s;
    std::back_insert_iterator<std::string> out(s);
    if (boost::spirit::karma::generate_delimited(out, g, d, attr) && s == expected)
        std::cout << "ok" << std::endl;
    else
        std::cout << "fail" << std::endl;
}
```

The examples of the binary generators use one or more of the following tests.
template <typename G>
void test_binary_generator(char const* expected, std::size_t size, G const& g)
{
    std::string s;
    std::back_insert_iterator<std::string> out(s);
    if (boost::spirit::karma::generate(out, g) && !std::memcmp(s.c_str(), expected, size))
        std::cout << "ok" << std::endl;
    else
        std::cout << "fail" << std::endl;
}

template <typename G, typename T>
void test_binary_generator_attr(char const* expected, std::size_t size, G const& g, T const& attr)
{
    std::string s;
    std::back_insert_iterator<std::string> out(s);
    if (boost::spirit::karma::generate(out, g, attr) && !std::memcmp(s.c_str(), expected, size))
        std::cout << "ok" << std::endl;
    else
        std::cout << "fail" << std::endl;
}

Models
Predefined models include:

- any literal string, e.g. "Hello, World",
- a pointer/reference to a null-terminated array of characters
- a std::basic_string<Char>

The namespace boost::spirit::traits is open for users to provide their own specializations. The customization points implemented by Spirit.Karma usable to customize the behavior of generators are described in the section Customization of Attribute Handling.

Generator API

Iterator Based Generator API

Description
The library provides a couple of free functions to make generating a snap. These generator functions have two forms. The first form, generate, concatenates the output generated by the involved components without inserting any output in between. The second generate_delimited intersperses the output generated by the involved components using the given delimiter generator. Both versions can take in attributes by (constant) reference that hold the attribute values to output.

Header

```
// forwards to <boost/spirit/home/karma/generate.hpp>
#include <boost/spirit/home/karma/generate.hpp>
```

For variadic attributes:
The variadic attributes version of the API allows one or more attributes to be passed into the `generate` functions. The functions taking two or more attributes are usable when the generator expression is a `Sequence (<<)` only. In this case each of the attributes passed have to match the corresponding part of the sequence.

For the API functions deducing the correct (matching) generator type from the supplied attribute type:

Also, see Include Structure.

**Namespace**

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boost::spirit::karma::generate</code></td>
</tr>
<tr>
<td><code>boost::spirit::karma::generate_delimited</code></td>
</tr>
<tr>
<td><code>boost::spirit::karma::delimit_flag::predelimit</code></td>
</tr>
<tr>
<td><code>boost::spirit::karma::delimit_flag::dont_predelimit</code></td>
</tr>
</tbody>
</table>
Synopsis

```cpp
namespace boost { namespace spirit { namespace karma {

    template <typename OutputIterator, typename Expr>
    inline bool generate(
        OutputIterator& sink
        , Expr const& expr);

    template <typename OutputIterator, typename Expr, typename Attr1, typename Attr2, ..., typename AttrN>
    inline bool generate(
        OutputIterator& sink
        , Expr const& expr
        , Attr1 const& attr1, Attr2 const& attr2, ..., AttrN const& attrN);

    template <typename OutputIterator, typename Expr, typename Delimiter>
    inline bool generate_delimited(
        OutputIterator& sink
        , Expr const& expr
        , Delimiter const& delimiter
        , BOOST_SCOPED_ENUM(delimit_flag) pre_delimit = delimit_flag::dont_predelimit);

    template <typename OutputIterator, typename Expr, typename Delimiter, typename Attr1, typename Attr2, ..., typename AttrN>
    inline bool generate_delimited(
        OutputIterator& sink
        , Expr const& expr
        , Delimiter const& delimiter
        , Attr1 const& attr1, Attr2 const& attr2, ..., AttrN const& attrN);

}}}
```

**Note**

Starting with *Spirit* V2.5 (distributed with Boost V1.47) the placeholder `_val` can be used in semantic actions attached to top level generator components. In this case `_val` refers to the supplied attribute as a whole. For API functions taking more than one attribute argument `_val` will refer to a Fusion vector or references to the attributes.

*Spirit.Karma* generator API functions based on the automatic creation of the matching generator type:
namespace boost { namespace spirit { namespace karma
{
    template <typename OutputIterator, typename Attr, typename Delimiter>
    inline bool
    generate_delimited(
        OutputIterator& sink,
        Attr const& attr,
        Delimiter const& delimiter,
        BOOST_SCOPED_ENUM(delimit_flag) pre_delimit = delimit_flag::dont_predelimit);

    template <typename OutputIterator, typename Attr>
    inline bool
    generate(
        OutputIterator& sink,
        Attr const& attr);
}}}

All functions above return true if none of the involved generator components failed, and false otherwise. If during the process of the output generation the underlying output stream reports an error, the return value will be false as well.

The maximum number of supported arguments is limited by the preprocessor constant SPIRIT_ARGUMENTS_LIMIT. This constant defaults to the value defined by the preprocessor constant PHOENIX_LIMIT (which in turn defaults to 10).

Note

The variadic functions with two or more attributes internally combine (constant) references to all passed attributes into a fusion::vector and forward this as a combined attribute to the corresponding function taking one attribute.

The generate_delimited functions not taking an explicit delimit_flag as one of their arguments don't invoke the passed delimiter before starting to generate output from the generator expression. This can be enabled by using the other version of that function while passing delimit_flag::predelimit to the corresponding argument.

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputIterator</td>
<td>OutputIterator receiving the generated output.</td>
</tr>
<tr>
<td>Expr</td>
<td>An expression that can be converted to a Karma generator.</td>
</tr>
<tr>
<td>Delimiter</td>
<td>Generator used to delimit the output of the expression components.</td>
</tr>
<tr>
<td>Attr</td>
<td>An attribute type utilized to create the corresponding generator type from.</td>
</tr>
<tr>
<td>Attr1, Attr2, ..., AttrN</td>
<td>One or more attributes.</td>
</tr>
</tbody>
</table>

Stream Based Generator API

Description

The library provides a couple of Standard IO Manipulators allowing to integrate Spirit.Karma output generation facilities with Standard output streams. These generator manipulators have two forms. The first form, format, concatenates the output generated by the involved components without inserting any output in between. The second, format_delimited, intersperses the output generated by the involved components using the given delimiter generator. Both versions can take in attributes by (constant) reference that hold the attribute values to output.
Header

```
// forwards to <boost/spirit/home/karma/stream/format_manip.hpp>
#include <boost/spirit/include/karma_format.hpp>
```

For variadic attributes:

```
// forwards to <boost/spirit/home/karma/stream/format_manip_attr.hpp>
#include <boost/spirit/include/karma_format_attr.hpp>
```

The variadic attributes version of the API allows one or more attributes to be passed into the format manipulators. The manipulators taking two or more attributes are usable when the generator expression is a Sequence (<<) only. In this case each of the attributes passed have to match the corresponding part of the sequence.

For the API functions deducing the correct (matching) generator type from the supplied attribute type:

```
// forwards to <boost/spirit/home/karma/format_auto.hpp>
#include <boost/spirit/include/karma_format_auto.hpp>
```

Also, see Include Structure.

Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::karma::format</td>
</tr>
<tr>
<td>boost::spirit::karma::format_delimited</td>
</tr>
<tr>
<td>boost::spirit::karma::delimit_flag::predelimit</td>
</tr>
<tr>
<td>boost::spirit::karma::delimit_flag::dont_predelimit</td>
</tr>
</tbody>
</table>
Synopsis

```cpp
namespace boost { namespace spirit { namespace karma {

    template <typename Expr>
    inline <unspecified>
    format(Expr const& xpr);

    template <typename Expr,
              typename Attr1, typename Attr2, ..., typename AttrN>
    inline <unspecified>
    format(Expr const& xpr,
           Attr1 const& attr1, Attr2 const& attr2, ..., AttrN const& attrN);

    template <typename Expr, typename Delimiter>
    inline <unspecified>
    format_delimited(Expr const& expr,
                     Delimiter const& d,
                     BOOST_SCOPED_ENUM(delimit_flag) pre_delimit = delimit_flag::dont_predelimit);

    template <typename Expr, typename Delimiter,
              typename Attr1, typename Attr2, ..., typename AttrN>
    inline <unspecified>
    format_delimited(Expr const& expr,
                     Delimiter const& d,
                     Attr1 const& attr1, Attr2 const& attr2, ..., AttrN const& attrN);

}}}
```

Spirit.Karma generator API functions based on the automatic creation of the matching generator type:

```cpp
namespace boost { namespace spirit { namespace karma {

    template <typename Attr, typename Delimiter>
    inline <unspecified>
    format_delimited(Attr const& attr,
                     Delimiter const& d,
                     BOOST_SCOPED_ENUM(delimit_flag) pre_delimit = delimit_flag::dont_predelimit);

}}}
```

All functions above return a standard IO stream manipulator instance (see Manipulators), which when streamed to an output stream will result in generating the output as emitted by the embedded Spirit.Karma generator expression. Any error occurring during the invocation of the Spirit.Karma generators will be reflected in the streams status flag (std::ios_base::failbit will be set).
The maximum number of supported arguments is limited by the preprocessor constant SPIRIT_ARGUMENTS_LIMIT. This constant defaults to the value defined by the preprocessor constant PHOENIX_LIMIT (which in turn defaults to 10).

**Note**

The variadic manipulators with two or more attributes internally combine (constant) references to all passed attributes into a `fusion::vector` and forward this as a combined attribute to the corresponding manipulator taking one attribute.

The `format_delimited` manipulators not taking an explicit `delimit_flag` as one of their arguments don't invoke the passed delimiter before starting to generate output from the generator expression. This can be enabled by using the other version of that manipulator while passing `delimit_flag::predelimit` to the corresponding argument.

**Template parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>An expression that can be converted to a Karma generator.</td>
</tr>
<tr>
<td>Delimiter</td>
<td>Generator used to delimit the output of the expression components.</td>
</tr>
<tr>
<td>Attr</td>
<td>An attribute type utilized to create the corresponding generator type from.</td>
</tr>
<tr>
<td>Attr1, Attr2, ..., AttrN</td>
<td>One or more attributes.</td>
</tr>
</tbody>
</table>

**API for Automatic Generator Creation**

**Description**

The library implements a special API returning a generator instance for a supplied attribute type. This function finds the best matching generator type for the attribute based on a set of simple matching rules (as outlined in the table below) applied recursively to the attribute type. The returned generator can be utilized to emit output for the provided attribute.

**Header**

```cpp
// forwards to <boost/spirit/home/karma/auto.hpp>
#include <boost/spirit/include/karma_auto.hpp>
```

Also, see Include Structure.

**Namespace**

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::karma::create_generator</td>
</tr>
<tr>
<td>boost::spirit::traits::create_generator_exists</td>
</tr>
</tbody>
</table>
## Synopsis

```cpp
namespace boost { namespace spirit { namespace karma
{
    template <typename Attr>
    inline <unspecified>
    create_generator();
}}}
```

The returned instance can be directly passed as the generator (or the delimiting generator) to any of the *Spirit.Karma* API functions. Additionally it can be assigned to a rule as the rules right hand side expression. This function will return a valid generator type only if the meta function `traits::create_generator_exists` returns `mpl::true_`. Otherwise it will fail compiling.

```cpp
namespace boost { namespace spirit { namespace traits
{
    template <typename Attr>
    struct create_generator_exists;
}}}
```

The meta function evaluates to `mpl::true_` if `create_generator` would return a valid generator for the given type `Attr`.

The following table outlines the mapping rules from the attribute type to the generator type. These rules are applied recursively to create the generator type which can be used to generate output from the given attribute type.

<table>
<thead>
<tr>
<th>Attribute type</th>
<th>Generator type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, wchar_t</td>
<td><code>std::char</code>, <code>std::wchar_t</code></td>
</tr>
<tr>
<td>short, int, long</td>
<td><code>std::short</code>, <code>std::int</code>, <code>std::long</code></td>
</tr>
<tr>
<td>unsigned short, unsigned int, unsigned long</td>
<td><code>std::ushort</code>, <code>std::uint</code>, <code>std::ulong</code></td>
</tr>
<tr>
<td>float, double, long double</td>
<td><code>std::float</code>, <code>std::double</code>, <code>std::longdouble</code></td>
</tr>
<tr>
<td>short, int, long</td>
<td><code>std::short</code>, <code>std::int</code>, <code>std::long</code></td>
</tr>
<tr>
<td>long long, unsigned long long</td>
<td><code>std::longlong</code>, <code>std::ulonglong</code></td>
</tr>
<tr>
<td>bool</td>
<td><code>std::bool</code></td>
</tr>
<tr>
<td>Any string (char const*, <code>std::string</code>, etc.)</td>
<td><code>std::string</code></td>
</tr>
<tr>
<td>Any (STL) container</td>
<td>Kleene Star (unary <code>'*'</code>)</td>
</tr>
<tr>
<td>Any Fusion sequence</td>
<td>Sequence operator (<code>'&lt;&lt;'</code>)</td>
</tr>
<tr>
<td><code>boost::optional&lt;&gt;</code></td>
<td>Optional operator (unary <code>'-'</code>)</td>
</tr>
<tr>
<td><code>boost::variant&lt;&gt;</code></td>
<td>Alternative operator (`'</td>
</tr>
</tbody>
</table>

### Important

The mapping for the generators `long long` and `ulong long` are only available on platforms where the preprocessor constant `BOOST_HAS_LONG_LONG` is defined (i.e. on platforms having native support for `long long` and `unsigned long long` (64 bit) signed and unsigned integer types).
Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attr</td>
<td>An attribute type utilized to create the corresponding generator type from.</td>
</tr>
</tbody>
</table>

Semantic Actions with Generators

Description

Semantic actions may be attached to any point in the grammar specification. They allow to call a function or function object in order to provide the value output by the generator attached to the semantic action. Semantic actions are associated with a generator using the syntax \( g[fa] \), where \( g \) is an arbitrary generator expression.

Header

```cpp
// forwards to <boost/spirit/home/karma/action.hpp>
#include <boost/spirit/include/karma_action.hpp>
```

Also, see Include Structure.

Model of

UnaryGenerator

Notation

\( a, g \) Instances of a generator, \( G \)

\( A \) Attribute type exposed by a generator, \( a \)

\( fa \) A (semantic action) function with signature \( \text{void}(\text{Attrib}&, \text{Context}&, \text{bool}&) \). The third parameter is a boolean flag that can be set to false to force the generator to fail. Both \( \text{Context} \) and the boolean flag are optional. For more information see below.

\( \text{Attrib} \) The attribute to be used to generate output from.

\( \text{Context} \) The type of the generator execution context. For more information see below.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryGenerator.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g[fa] )</td>
<td>Call semantic action, ( fa ) before invoking ( g ). The function or function object ( fa ) is expected to provide the value to generate output from to the generator ( g ).</td>
</tr>
</tbody>
</table>

The possible signatures for functions to be used as semantic actions are:
The function or function object is expected to return the value to generate output from by assigning it to the first parameter, \texttt{attr}. Here \texttt{Attrib} is the attribute type of the generator attached to the semantic action.

The type \texttt{Context} is the type of the generator execution context. This type is unspecified and depends on the context the generator is invoked in. The value \texttt{context} is used by semantic actions written using Phoenix to access various context dependent attributes and values. For more information about Phoenix placeholder expressions usable in semantic actions see \texttt{Nonterminal}.

The third parameter, \texttt{pass}, can be used by the semantic action to force the associated generator to fail. If \texttt{pass} is set to \texttt{false} the action generator will immediately return \texttt{false} as well, while not invoking \texttt{g} and not generating any output.

\textbf{Attributes}

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{a[fa]}</td>
<td>\texttt{a: A --&gt; a[fa]: A}</td>
</tr>
</tbody>
</table>

\textbf{Complexity}

The complexity of the action generator is defined by the complexity of the generator the semantic action is attached to and the complexity of the function or function object used as the semantic action.

\textbf{Important}

Please note that the use of semantic actions in \texttt{Spirit.Karma} generally forces the library to create a \texttt{copy} of the attribute, which might be a costly operation. Always consider using other means of associating a value with a particular generator first.

\textbf{Example}

\textbf{Note}

The test harness for the example(s) below is presented in the \texttt{Basics Examples} section.

Some includes:

\begin{verbatim}
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
\end{verbatim}

Some using declarations:
using boost::spirit::karma::int_;  
using boost::spirit::karma::string;  
using boost::spirit::karma::_1;  
using boost::phoenix::ref;  
using boost::phoenix::val;

Some examples:

```cpp
int i = 42;
test_generator("42", int_[_1 = ref(i)]);
test_generator("abc", string[_1 = val("abc")]);
```

More examples for semantic actions can be found here: Examples of Semantic Actions.

**Auto Generator**

**Description**

This module includes the description of the `auto_` generator. This generator can be used to automatically create a generator based on the supplied attribute type.

**Header**

```
// forwards to <boost/spirit/home/karma/auto.hpp>  
#include <boost/spirit/include/karma_auto.hpp>
```

Also, see Include Structure.

**Namespace**

**Name**

```
boost::spirit::auto_  // alias: boost::spirit::karma::auto_
```

**Model of**

- PrimitiveGenerator

**Notation**

A variable instance of any type for which a mapping to a generator type is defined (the meta function `traits::create_generator_exists` returns `mpl::true_`) or a Lazy Argument that evaluates to any type for which a mapping to a generator type is defined (the meta function `traits::create_generator_exists` returns `mpl::true_`).

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveGenerator.
Create a generator instance compatible with the supplied attribute type and use it for output generation. This generator never fails (unless the underlying output stream reports an error).

Create a generator instance compatible with the supplied literal value. This generator never fails (unless the underlying output stream reports an error).

**Additional Requirements**

The `auto_` generators can be used to emit output for any data type for which a mapping to a generator type is defined (the metafunction `traits::create_generator_exists` returns `mpl::true_`). The following table outlines the predefined mapping rules from the attribute type to the generator type. These rules are applied recursively to create the generator type which can be used to generate output from the given attribute type.

<table>
<thead>
<tr>
<th>Attribute type</th>
<th>Generator type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, wchar_t</td>
<td>standard::char_, standard::char_</td>
</tr>
<tr>
<td>short, int, long</td>
<td>short_, int_, long_</td>
</tr>
<tr>
<td>unsigned short, unsigned int, unsigned long</td>
<td>unsigned short_, unsigned int_, ulong_</td>
</tr>
<tr>
<td>float, double, long double</td>
<td>float_, double_, long_, double</td>
</tr>
<tr>
<td>short, int, long</td>
<td>short_, int_, long_</td>
</tr>
<tr>
<td>long long, unsigned long long</td>
<td>long_long, ulong_long</td>
</tr>
<tr>
<td>bool</td>
<td>bool_</td>
</tr>
<tr>
<td>Any string (char const*, std::string, etc.)</td>
<td>string</td>
</tr>
<tr>
<td>Any (STL) container</td>
<td>Kleene Star (unary '*')</td>
</tr>
<tr>
<td>Any Fusion sequence</td>
<td>Sequence operator ('&lt;&lt;')</td>
</tr>
<tr>
<td>boost::optional&lt;&gt;</td>
<td>Optional operator (unary '-')</td>
</tr>
<tr>
<td>boost::variant&lt;&gt;</td>
<td>Alternative operator ('</td>
</tr>
</tbody>
</table>

It is possible to add support for any custom data type by implementing a specialization of the customization point `traits::create_generator`. This customization can be used also to redefined any of the predefined mappings.

**Attributes**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto_</td>
<td>hold_any, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>auto_(s)</td>
<td>unused</td>
</tr>
</tbody>
</table>
Important

The attribute type hold\_any exposed by some of the auto\_ generators is semantically and syntactically equivalent to the type implemented by Boost.Any. It has been added to Spirit as it has better a performance and a smaller footprint if compared to Boost.Any.

Note

In addition to their usual attribute of type Attrib all listed generators accept an instance of a boost::optional<Attrib> as well. If the boost::optional<> is initialized (holds a value) the generators behave as if their attribute was an instance of Attrib and emit the value stored in the boost::optional<>. Otherwise the generators will fail.

Complexity

The complexity of the auto\_ generator depends on the attribute type. Each attribute type results in a different generator type to be instantiated which defines the overall complexity.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::auto_;  
```

And a class definition used in the examples:

```cpp
// a simple complex number representation z = a + bi
struct complex
{
    complex (double a, double b)
    : a(a), b(b)
    {}

    double a;
    double b;
};
```

The following construct is required to allow the complex data structure to be utilized as a Boost.Fusion sequence. This is required as we will emit output for this data structure with a Spirit.Karma sequence: `'|' << karma::double_ << ','' << karma::double_ << '}'.

257
We add a specialization for the `create_generator` customization point defining a custom output format for the complex type. Generally, any specialization for `create_generator` is expected to return the proto expression to be used to generate output for the type the customization point has been specialized for.

We need to utilize `proto::deep_copy` as the expression contains literals (the '{', ',', '}', and '}') which normally get embedded in the proto expression by reference only. The deep copy converts the proto tree to hold this by value. The deep copy operation can be left out for simpler proto expressions (not containing references to temporaries). Alternatively you could use the `proto::make_expr` facility to build the required proto expression.

```cpp
namespace boost { namespace spirit { namespace traits {
    template <>
    struct create_generator<complex> {
        typedef proto::result_of::deep_copy<
            BOOST_TYPEOF('{' << karma::double_ << ',' << karma::double_ << '}'
        )::type type;

        static type call() {
            return proto::deep_copy(
                '{' << karma::double_ << ',' << karma::double_ << '}'
            );
        }
    };
}}}
```

Some usage examples of `auto_` generators:

Emit a simple string using the `karma::string` generator:

```cpp
test_generator_attr("abc", auto_, "abc");
test_generator("abc", auto_("abc"));
```

Emit instances of the `complex` data type as defined above using the generator defined by the customization point for `complex`:

```cpp
test_generator_attr("{1.2,2.4}", auto_, complex(1.2, 2.4));
test_generator("{1.2,2.4}", auto_(complex(1.2, 2.4)));
```

### Auxiliary Generators

This module includes different auxiliary generators not fitting into any of the other categories. It includes the `attr_cast, eol, eps, and lazy` generators.

#### Module Header

```cpp
// forwards to <boost/spirit/home/karma/auxiliary.hpp>
#include <boost/spirit/include/karma_auxiliary.hpp>
```

Also, see Include Structure.
**Attribute Transformation Pseudo Generator (attr_cast)**

**Description**

The `attr_cast<Exposed, Transformed>()` component invokes the embedded generator while supplying an attribute of type `Transformed`. The supplied attribute gets created from the original attribute (of type `Exposed`) passed to this component using the customization point `traits::transform_attribute`.

**Header**

```cpp
// forwards to <boost/spirit/home/karma/auxiliary/attr_cast.hpp>
#include <boost/spirit/include/karma_attr_cast.hpp>
```

Also, see Include Structure.

**Namespace**

**Name**

```cpp
boost::spirit::attr_cast // alias: boost::spirit::karma::attr_cast
```

**Synopsis**

```cpp
template <Exposed, Transformed>
<unspecified> attr_cast(<unspecified>);
```

**Template parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exposed</td>
<td>The type of the attribute supplied to the attr_cast.</td>
<td>unused_type</td>
</tr>
<tr>
<td>Transformed</td>
<td>The type of the attribute expected by the embedded generator g.</td>
<td>unused_type</td>
</tr>
</tbody>
</table>

The `attr_cast` is a function template. It is possible to invoke it using the following schemes:

```cpp
attr_cast(g)
attr_cast<Exposed>(g)
attr_cast<Exposed, Transformed>(g)
```

depending on which of the attribute types can be deduced properly if not explicitly specified.

**Model of**

UnaryGenerator

**Notation**

`g` A generator object.

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in UnaryGenerator.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr_cast (g)</td>
<td>Create a component invoking the generator g while passing an attribute of the type as normally expected by g. The type of the supplied attribute will be transformed to the type g exposes as its attribute type (by using the attribute customization point traits::transform_attribute). This generator does not fail unless g fails.</td>
</tr>
<tr>
<td>attr_cast&lt;Exposed&gt; (g)</td>
<td>Create a component invoking the generator g while passing an attribute of the type as normally expected by g. The supplied attribute is expected to be of the type Exposed, it will be transformed to the type g exposes as its attribute type (using the attribute customization point traits::transform_attribute). This generator does not fail unless g fails.</td>
</tr>
<tr>
<td>attr_cast&lt;Exposed, Transformed&gt; (g)</td>
<td>Create a component invoking the generator g while passing an attribute of type Transformed. The supplied attribute is expected to be of the type Exposed, it will be transformed to the type Transformed (using the attribute customization point traits::transform_attribute). This generator does not fail unless g fails.</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr_cast (g)</td>
<td>g: A --&gt; attr_cast (g): A</td>
</tr>
<tr>
<td>attr_cast&lt;Exposed&gt; (g)</td>
<td>g: A --&gt; attr_cast&lt;Exposed&gt; (g): Exposed</td>
</tr>
<tr>
<td>attr_cast&lt;Exposed, Transformed&gt; (g)</td>
<td>g: A --&gt; attr_cast&lt;Exposed, Transformed&gt; (g): Exposed</td>
</tr>
</tbody>
</table>

### Complexity

The complexity of this component is fully defined by the complexity of the embedded generator g.

### Example

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:
using boost::spirit::karma::int_;

The example references data structure `int_data` which needs a specialization of the customization point `traits::transform_attribute`:

```cpp
// this is just a test structure we want to use in place of an int
struct int_data
{
    int i;
};

// we provide a custom attribute transformation to allow its use as an int
namespace boost { namespace spirit { namespace traits
{
    template <>
    struct transform_attribute<int_data const, int, karma::domain>
    {
        typedef int type;
        static int pre(int_data const& d) { return d.i; }
    };  
}}}
```

Now we use the `attr_cast` pseudo generator to invoke the attribute transformation:

```cpp
int_data d = { 1 };  
test_generator_attr("1", boost::spirit::karma::attr_cast(int_), d);
```

### End of Line Generator (eol)

**Description**

The `eol` component generates a single newline character. It is equivalent to `lit(\n)` or simply `\n` (please see the `char_` generator module for more details).

**Header**

```cpp
// forwards to <boost/spirit/home/karma/auxiliary/eol.hpp>
#include <boost/spirit/include/karma_eol.hpp>
```

Also, see Include Structure.

**Namespace**

**Name**

`boost::spirit::eol // alias: boost::spirit::karma::eol`

**Model of**

`PrimitiveGenerator`

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in `PrimitiveGenerator`.  

261
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eol</code></td>
<td>Create a component generating a single end of line character in the output. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eol</code></td>
<td><code>unused</code></td>
</tr>
</tbody>
</table>

**Complexity**

O(1)

The complexity is constant as a single character is generated in the output.

**Example**

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::eol;
```

Basic usage of the `eol` generator:

```cpp
test_generator("\n", eol);
test_generator("abc\n", "abc" << eol);
```

**Epsilon Generator (`eps`)**

The family of `eps` components allows to create pseudo generators generating an empty string. This feature is sometimes useful either to force a generator to fail or to succeed or to insert semantic actions into the generation process.

**Description**

The Epsilon (`eps`) is a multi-purpose generator that emits a zero length string.

**Simple Form**

In its simplest form, `eps` creates a component generating an empty string while always succeeding:
eps  // always emits a zero-length string

This form is usually used to trigger a semantic action unconditionally. For example, it is useful in triggering error messages when a set of alternatives fail:

```
r = a | b | c | eps[error()]; // Call error if a, b, and c fail to generate
```

**Semantic Predicate**

The $\text{eps}(b)$ component generates an empty string as well, but succeeds only if $b$ is `true` and fails otherwise. Its lazy variant $\text{eps}(fb)$ is equivalent to $\text{eps}(b)$ except it evaluates the supplied function $fb$ at generate time, while using the return value as the criteria to succeed.

Semantic predicates allow you to attach a conditional function anywhere in the grammar. In this role, the epsilon takes a Lazy Argument that returns `true` or `false`. The Lazy Argument is typically a test that is called to resolve ambiguity in the grammar. A generator failure will be reported when the Lazy Argument result evaluates to `false`. Otherwise an empty string will be emitted. The general form is:

```
eps_p(fb) << rest;
```

The Lazy Argument $fb$ is called to do a semantic test. If the test returns true, $rest$ will be evaluated. Otherwise, the production will return early without ever touching rest.

**Header**

```
// forwards to <boost/spirit/home/karma/auxiliary/eps.hpp>
#include <boost/spirit/include/karma_eps.hpp>
```

Also, see Include Structure.

**Namespace**

**Name**

```
boost::spirit::eps // alias: boost::spirit::karma::eps
```

**Model of**

PrimitiveGenerator

**Notation**

- $b$ A boolean value.
- $fb$ A Lazy Argument that evaluates to a boolean value.

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveGenerator.
### Expression | Semantics
--- | ---
eps | Creates a component generating an empty string. Succeeds always.
eps(b) | Creates a component generating an empty string. Succeeds if \( b \) is \( \text{true} \) (unless the underlying output stream reports an error).
eps(fb) | Creates a component generating an empty string. Succeeds if \( fb \) returns \( \text{true} \) at generate time (unless the underlying output stream reports an error).

### Attributes

| Expression | Attribute |
--- | ---
eps | unused |
eps(b) | unused |
eps(fb) | unused |

### Complexity

\( O(1) \)
The complexity is constant as no output is generated.

### Example

#### Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::eps;
using boost::phoenix::val;
```

Basic usage of the \( \text{eps} \) generator:
Lazy Generator (**lazy**)

**Description**

The family of *lazy* components allows to use a dynamically returned generator component for output generation. It calls the provided function or function object at generate time using its return value as the actual generator to produce the output.

**Header**

```
// forwards to <boost/spirit/home/karma/auxiliary/lazy.hpp>
#include <boost/spirit/include/karma_lazy.hpp>
```

Also, see Include Structure.

**Namespace**

```c++
boost::spirit::lazy // alias: boost::spirit::karma::lazy
```

**Model of**

*Generator*

**Notation**

fg  A function or function object that evaluates to a generator object (an object exposing the *Generator*). This function will be invoked at generate time.

The signature of `fg` is expected to be

```
G f(Unused, Context)
```

where `G`, the function's return value, is the type of the generator to be invoked, and `Context` is the generator's Context type (The first argument is `unused` to make the `Context` the second argument. This is done for uniformity with *Semantic Actions*).

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in *Generator*. 

```c++
test_generator("abc", eps[std::cout << val("starting eps example")]) << "abc");
test_generator("abc", eps(true) << "abc");
test_generator("", eps(false) << "abc");  // fails as eps expression is 'false'
```
The Boost.Phoenix function object \( fg \) will be invoked at generate time. It is expected to return a generator instance. This generator is then invoked in order to generate the output. This generator will succeed as long as the invoked generator succeeds as well (unless the underlying output stream reports an error).

The function or function object will be invoked at generate time. It is expected to return a generator instance (note this version of lazy does not require \( fg \) to be a Boost.Phoenix function object). This generator is then invoked in order to generate the output. This generator will succeed as long as the invoked generated succeeds as well (except if the underlying output stream reports an error).

### Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>( fg )</td>
<td>The attribute type ( G ) as exposed by the generator ( g ) returned from ( fg ).</td>
</tr>
<tr>
<td>lazy(( fg ))</td>
<td>The attribute type ( G ) as exposed by the generator ( g ) returned from ( fg ).</td>
</tr>
</tbody>
</table>

### Complexity

The complexity of the lazy component is determined by the complexity of the generator returned from \( fg \).

### Example

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
namespace karma = boost::spirit::karma;
using boost::spirit::karma::_1;
using boost::spirit::ascii::string;
using boost::phoenix::val;
```

Basic usage of the lazy generator:
Binary Generators

This module includes different generators allowing to output binary data. It includes generators for default, little, and big endian binary output and a pad generator allowing to control padding of the generated output stream.

Module Header

```cpp
// forwards to <boost/spirit/home/karma/binary.hpp>
#include <boost/spirit/include/karma_binary.hpp>
```

Also, see Include Structure.

Binary Native Endianness Generators

Description

The binary native endianness generators described in this section are used to emit binary byte streams laid out conforming to the native endianness (byte order) of the target architecture.

Header

```cpp
// forwards to <boost/spirit/home/karma/binary.hpp>
#include <boost/spirit/include/karma_binary.hpp>
```

Also, see Include Structure.

Namespace

```cpp
Name

boost::spirit::byte_ // alias: boost::spirit::karma::byte_
boost::spirit::word // alias: boost::spirit::karma::word
boost::spirit::dword // alias: boost::spirit::karma::dword
boost::spirit::qword // alias: boost::spirit::karma::qword
```

Note

The generators qword and qword(qw) are only available on platforms where the preprocessor constant BOOST_HAS_LONG_LONG is defined (i.e. on platforms having native support for unsigned long long (64 bit) integer types).

Model of

PrimitiveGenerator

Notation

- A single byte (8 bit binary value) or a Lazy Argument that evaluates to a single byte
A 16 bit binary value or a Lazy Argument that evaluates to a 16 bit binary value. This value is always interpreted using native endianness.

dw A 32 bit binary value or a Lazy Argument that evaluates to a 32 bit binary value. This value is always interpreted using native endianness.

qw A 64 bit binary value or a Lazy Argument that evaluates to a 64 bit binary value. This value is always interpreted using native endianness.

### Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveGenerator.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte_</td>
<td>Output the binary representation of the least significant byte of the mandatory attribute. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>word</td>
<td>Output the binary representation of the least significant 16 bits of the mandatory attribute in native endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>dword</td>
<td>Output the binary representation of the least significant 32 bits of the mandatory attribute in native endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>qword</td>
<td>Output the binary representation of the least significant 64 bits of the mandatory attribute in native endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>byte_ (b)</td>
<td>Output the binary representation of the least significant byte of the immediate parameter. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>word (w)</td>
<td>Output the binary representation of the least significant 16 bits of the immediate parameter in native endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>dword (dw)</td>
<td>Output the binary representation of the least significant 32 bits of the immediate parameter in native endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>qword (qw)</td>
<td>Output the binary representation of the least significant 64 bits of the immediate parameter in native endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
</tbody>
</table>
### Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte_</td>
<td>boost::uint_least8_t, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>word</td>
<td>boost::uint_least16_t, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>dword</td>
<td>boost::uint_least32_t, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>qword</td>
<td>boost::uint_least64_t, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>byte_(b)</td>
<td>unused</td>
</tr>
<tr>
<td>word(w)</td>
<td>unused</td>
</tr>
<tr>
<td>dword(dw)</td>
<td>unused</td>
</tr>
<tr>
<td>qword(qw)</td>
<td>unused</td>
</tr>
</tbody>
</table>

#### Note

In addition to their usual attribute of type `Attrib` all listed generators accept an instance of a `boost::optional<Attrib>` as well. If the `boost::optional<>` is initialized (holds a value) the generators behave as if their attribute was an instance of `Attrib` and emit the value stored in the `boost::optional<.>` otherwise the generators will fail.

### Complexity

$O(N)$, where $N$ is the number of bytes emitted by the binary generator

### Example

#### Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:
using boost::spirit::karma::byte_;  
using boost::spirit::karma::word;  
using boost::spirit::karma::dword;  
using boost::spirit::karma::qword;  

Basic usage of the native binary generators with some results for little endian platforms:

test_binary_generator("\x01", 1, byte_(0x01));  
test_binary_generator("\x01\x02", 2, word(0x0201));  
test_binary_generator("\x01\x02\x03\x04", 4, dword(0x04030201));  
test_binary_generator("\x01\x02\x03\x04\x05\x06\x07\x08", 8, qword(0x0807060504030201LL));  

Basic usage of the native binary generators with some results for big endian platforms:

test_binary_generator("\x01", 1, byte_(0x01));  
test_binary_generator("\x02\x01", 2, word(0x0201));  
test_binary_generator("\x04\x03\x02\x01", 4, dword(0x04030201));  
test_binary_generator("\x08\x07\x06\x05\x04\x03\x02\x01", 8, qword(0x0807060504030201LL));  

Binary Little Endianness Generators

Description

The little native endianness generators described in this section are used to emit binary byte streams laid out conforming to the little endianness byte order.

Header

// forwards to <boost/spirit/home/karma/binary.hpp>  
#include <boost/spirit/include/karma_binary.hpp>  

Also, see Include Structure.

Namespace

Name

boost::spirit::little_word // alias: boost::spirit::karma::little_word  
boost::spirit::little_dword // alias: boost::spirit::karma::little_dword  
boost::spirit::little_qword // alias: boost::spirit::karma::little_qword
Note

The generators `little_qword` and `little_qword(qw)` are only available on platforms where the preprocessor constant `BOOST_HAS_LONG_LONG` is defined (i.e. on platforms having native support for `unsigned long long` (64 bit) integer types).

Model of

`PrimitiveGenerator`

Notation

`w` A 16 bit binary value or a Lazy Argument that evaluates to a 16 bit binary value. This value is always interpreted using native endianness.

`dw` A 32 bit binary value or a Lazy Argument that evaluates to a 32 bit binary value. This value is always interpreted using native endianness.

`qw` A 64 bit binary value or a Lazy Argument that evaluates to a 64 bit binary value. This value is always interpreted using native endianness.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in `PrimitiveGenerator`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>little_word</code></td>
<td>Output the binary representation of the least significant 16 bits of the mandatory attribute in little endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>little_dword</code></td>
<td>Output the binary representation of the least significant 32 bits of the mandatory attribute in little endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>little_qword</code></td>
<td>Output the binary representation of the least significant 64 bits of the mandatory attribute in little endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>little_word(w)</code></td>
<td>Output the binary representation of the least significant 16 bits of the immediate parameter in little endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>little_dword(dw)</code></td>
<td>Output the binary representation of the least significant 32 bits of the immediate parameter in little endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>little_qword(qw)</code></td>
<td>Output the binary representation of the least significant 64 bits of the immediate parameter in little endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
</tbody>
</table>
**Attributes**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>little_word</td>
<td>boost::uint_least16_t, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>little_dword</td>
<td>boost::uint_least32_t, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>little_qword</td>
<td>boost::uint_least64_t, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>little_word(w)</td>
<td>unused</td>
</tr>
<tr>
<td>little_dword(dw)</td>
<td>unused</td>
</tr>
<tr>
<td>little_qword(qw)</td>
<td>unused</td>
</tr>
</tbody>
</table>

**Complexity**

O(N), where N is the number of bytes emitted by the binary generator

**Example**

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::little_word;
using boost::spirit::karma::little_dword;
using boost::spirit::karma::little_qword;
```

Basic usage of the little binary generators:
Binary Big Endianness Generators

Description

The big native endianness generators described in this section are used to emit binary byte streams laid out conforming to the big endianness byte order.

Header

```cpp
// forwards to <boost/spirit/home/karma/binary.hpp>
#include <boost/spirit/include/karma_binary.hpp>
```

Also, see Include Structure.

Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::big_word // alias: boost::spirit::karma::big_word</td>
</tr>
<tr>
<td>boost::spirit::big_dword // alias: boost::spirit::karma::big_dword</td>
</tr>
<tr>
<td>boost::spirit::big_qword // alias: boost::spirit::karma::big_qword</td>
</tr>
</tbody>
</table>

Note

The generators `big_qword` and `big_qword(qw)` are only available on platforms where the preprocessor constant `BOOST_HAS_LONG_LONG` is defined (i.e. on platforms having native support for `unsigned long long` (64 bit) integer types).

Model of

PrimitiveGenerator

Notation

- `w` A 16 bit binary value or a Lazy Argument that evaluates to a 16 bit binary value. This value is always interpreted using native endianness.
- `dw` A 32 bit binary value or a Lazy Argument that evaluates to a 32 bit binary value. This value is always interpreted using native endianness.
- `qw` A 64 bit binary value or a Lazy Argument that evaluates to a 64 bit binary value. This value is always interpreted using native endianness.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveGenerator.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>big_word</td>
<td>Output the binary representation of the least significant 16 bits of the mandatory attribute in big endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>big_dword</td>
<td>Output the binary representation of the least significant 32 bits of the mandatory attribute in big endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>big_qword</td>
<td>Output the binary representation of the least significant 64 bits of the mandatory attribute in big endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>big_word(w)</td>
<td>Output the binary representation of the least significant 16 bits of the immediate parameter in big endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>big_dword(dw)</td>
<td>Output the binary representation of the least significant 32 bits of the immediate parameter in big endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>big_qword(qw)</td>
<td>Output the binary representation of the least significant 64 bits of the immediate parameter in big endian representation. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>big_word</td>
<td>boost::uint_least16_t, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>big_dword</td>
<td>boost::uint_least32_t, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>big_qword</td>
<td>boost::uint_least64_t, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>big_word(w)</td>
<td>unused</td>
</tr>
<tr>
<td>big_dword(dw)</td>
<td>unused</td>
</tr>
<tr>
<td>big_qword(qw)</td>
<td>unused</td>
</tr>
</tbody>
</table>

**Complexity**

O(N), where N is the number of bytes emitted by the binary generator.
Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::big_word;
using boost::spirit::karma::big_dword;
using boost::spirit::karma::big_qword;
```

Basic usage of the big binary generators:

```cpp
test_binary_generator("\x02\x01", 2, big_word(0x0201));
test_binary_generator("\x04\x03\x02\x01", 4, big_dword(0x04030201));
test_binary_generator("\x08\x07\x06\x05\x04\x03\x02\x01", 8, big_qword(0x0807060504030201LL));
test_binary_generator_attr("\x02\x01", 2, big_word, 0x0201);
test_binary_generator_attr("\x04\x03\x02\x01", 4, big_dword, 0x04030201);
test_binary_generator_attr("\x08\x07\x06\x05\x04\x03\x02\x01", 8, big_qword, 0x0807060504030201LL);
```

Char Generators

This module includes different character oriented generators allowing to output single characters. Currently, it includes literal chars (e.g. 'x', 'x'), char_ (single characters, ranges and character sets) and the encoding specific character classifiers (alnum, alpha, digit, xdigit, etc.).

Module Header

```cpp
// forwards to <boost/spirit/home/karma/char.hpp>
#include <boost/spirit/include/karma_char.hpp>
```

Also, see Include Structure.

Character Generators (char_, lit)

Description

The character generators described in this section are:

The char_ generator emits single characters. The char_ generator has an associated Character Encoding Namespace. This is needed when doing basic operations such as forcing lower or upper case and dealing with character ranges.

There are various forms of char_.

275
char_

The no argument form of char_ emits any character in the associated Character Encoding Namespace.

```
chart_ // emits any character as supplied by the attribute
```

char_(ch)

The single argument form of char_ (with a character argument) emits the supplied character.

```
char_('x')    // emits 'x'
char_(L'x')  // emits L'x'
char_(x)     // emits x (a char)
```

char_(first, last)

char_ with two arguments, emits any character from a range of characters as supplied by the attribute.

```
char_('a','z')  // alphabetic characters
char_(L'0',L'9')  // digits
```

A range of characters is created from a low-high character pair. Such a generator emits a single character that is in the range, including both endpoints. Note, the first character must be before the second, according to the underlying Character Encoding Namespace.

Character mapping is inherently platform dependent. It is not guaranteed in the standard for example that 'A' < 'z', that is why in Spirit2, we purposely attach a specific Character Encoding Namespace (such as ASCII, ISO-8859-1) to the char_ generator to eliminate such ambiguities.

Note

Sparse bit vectors

To accommodate 16/32 and 64 bit characters, the char-set statically switches from a std::bitset implementation when the character type is not greater than 8 bits, to a sparse bit/boolean set which uses a sorted vector of disjoint ranges (range_run). The set is constructed from ranges such that adjacent or overlapping ranges are coalesced.

range_runs are very space-economical in situations where there are lots of ranges and a few individual disjoint values. Searching is O(log n) where n is the number of ranges.

char_(def)

Lastly, when given a string (a plain C string, a std::basic_string, etc.), the string is regarded as a char-set definition string following a syntax that resembles posix style regular expression character sets (except that double quotes delimit the set elements instead of square brackets and there is no special negation ^ character). Examples:

```
char_("a-zA-Z")    // alphabetic characters
char_("0-9a-fA-F") // hexadecimal characters
char_("actgACTG")  // DNA identifiers
char_("\x7f\x7e")  // Hexadecimal 0x7F and 0x7E
```

These generators emit any character from a range of characters as supplied by the attribute.

lit(ch)

lit, when passed a single character, behaves like the single argument char_ except that lit does not consume an attribute. A plain char or wchar_t is equivalent to a lit.
lit is reused by the String Generators, the char generators, and the Numeric Generators (see signed integer, unsigned integer, and real number generators). In general, a char generator is created when you pass in a character, a string generator is created when you pass in a string, and a numeric generator is created when you use a numeric literal. The exception is when you pass a single element literal string, e.g. `lit("x")`. In this case, we optimize this to create a char generator instead of a string generator.

Examples:

```
'x'
lit('x')
lit(L'x')
lit(c)    // c is a char
```

Header

```
// forwards to <boost/spirit/home/karma/char/char.hpp>
#include <boost/spirit/include/karma_char_.hpp>
```

Also, see Include Structure.

Namespace

```
#include <boost/spirit/include/karma_char_.hpp>
```

Also, see Include Structure.

Namespace

```
boost::spirit::lit // alias: boost::spirit::karma::lit
ns::char_
```

In the table above, `ns` represents a Character Encoding Namespace.

Model of

```
PrimitiveGenerator
```

Notation

```
ch, ch1, ch2  Character-class specific character (See Character Class Types), or a Lazy Argument that evaluates to a character-class specific character value

cs  Character-set specifier string (See Character Class Types), or a Lazy Argument that evaluates to a character-set specifier string, or a pointer/reference to a null-terminated array of characters. This string specifies a char-set definition string following a syntax that resembles posix style regular expression character sets (except the square brackets and the negation \^ character).

ns  A Character Encoding Namespace.

cg  A char generator, a char range generator, or a char set generator.
```

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveGenerator.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ch</code></td>
<td>Generate the character literal <code>ch</code>. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>lit(ch)</code></td>
<td>Generate the character literal <code>ch</code>. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>ns::char_</code></td>
<td>Generate the character provided by a mandatory attribute interpreted in the character set defined by <code>ns</code>. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>ns::char_(ch)</code></td>
<td>Generate the character <code>ch</code> as provided by the immediate literal value the generator is initialized from. If this generator has an associated attribute it succeeds only as long as the attribute is equal to the immediate literal (unless the underlying output stream reports an error). Otherwise this generator fails and does not generate any output.</td>
</tr>
<tr>
<td><code>ns::char_(&quot;c&quot;)</code></td>
<td>Generate the character <code>c</code> as provided by the immediate literal value the generator is initialized from. If this generator has an associated attribute it succeeds only as long as the attribute is equal to the immediate literal (unless the underlying output stream reports an error). Otherwise this generator fails and does not generate any output.</td>
</tr>
<tr>
<td><code>ns::char_(ch1, ch2)</code></td>
<td>Generate the character provided by a mandatory attribute interpreted in the character set defined by <code>ns</code>. The generator succeeds as long as the attribute belongs to the character range <code>[ch1, ch2]</code> (unless the underlying output stream reports an error). Otherwise this generator fails and does not generate any output.</td>
</tr>
<tr>
<td><code>ns::char_(cs)</code></td>
<td>Generate the character provided by a mandatory attribute interpreted in the character set defined by <code>ns</code>. The generator succeeds as long as the attribute belongs to the character set <code>cs</code> (unless the underlying output stream reports an error). Otherwise this generator fails and does not generate any output.</td>
</tr>
<tr>
<td><code>~cg</code></td>
<td>Negate <code>cg</code>. The result is a negated char generator that inverts the test condition of the character generator it is attached to.</td>
</tr>
</tbody>
</table>

A character `ch` is assumed to belong to the character range defined by `ns::char_(ch1, ch2)` if its character value (binary representation) interpreted in the character set defined by `ns` is not smaller than the character value of `ch1` and not larger then the character value of `ch2` (i.e. `ch1 <= ch <= ch2`).

The `charset` parameter passed to `ns::char_(charset)` must be a string containing more than one character. Every single character in this string is assumed to belong to the character set defined by this expression. An exception to this is the `'-'` character which has a special meaning if it is not specified as the first and not the last character in `charset`. If the `'-'` is used in between to characters it is interpreted as spanning a character range. A character `ch` is considered to belong to the defined character set `charset` if it matches one of the characters as specified by the string parameter described above. For example
Example | Description
---|---
char_("abc") | 'a', 'b', and 'c'
char_("a-z") | all characters (and including) from 'a' to 'z'
char_("a-zA-Z") | all characters (and including) from 'a' to 'z' and 'A' and 'Z'
char_("1-9") | '-' and all characters (and including) from '1' to '9'

Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>unused</td>
</tr>
<tr>
<td>lit(ch)</td>
<td>unused</td>
</tr>
<tr>
<td>ns::char_</td>
<td>Ch, attribute is mandatory (otherwise compilation will fail). Ch is the character type of the Character Encoding Namespace, ns.</td>
</tr>
<tr>
<td>ns::char_(ch)</td>
<td>Ch, attribute is optional, if it is supplied, the generator compares the attribute with ch and succeeds only if both are equal, failing otherwise. Ch is the character type of the Character Encoding Namespace, ns.</td>
</tr>
<tr>
<td>ns::char_(&quot;c&quot;)</td>
<td>Ch, attribute is optional, if it is supplied, the generator compares the attribute with c and succeeds only if both are equal, failing otherwise. Ch is the character type of the Character Encoding Namespace, ns.</td>
</tr>
<tr>
<td>ns::char_(ch1, ch2)</td>
<td>Ch, attribute is mandatory (otherwise compilation will fail), the generator succeeds if the attribute belongs to the character range [ch1, ch2] interpreted in the character set defined by ns. Ch is the character type of the Character Encoding Namespace, ns.</td>
</tr>
<tr>
<td>ns::char_(cs)</td>
<td>Ch, attribute is mandatory (otherwise compilation will fail), the generator succeeds if the attribute belongs to the character set cs, interpreted in the character set defined by ns. Ch is the character type of the Character Encoding Namespace, ns.</td>
</tr>
<tr>
<td>~cg</td>
<td>Attribute of cg</td>
</tr>
</tbody>
</table>

Note

In addition to their usual attribute of type Ch all listed generators accept an instance of a boost::optional<Ch> as well. If the boost::optional<> is initialized (holds a value) the generators behave as if their attribute was an instance of Ch and emit the value stored in the boost::optional<>. Otherwise the generators will fail.

Complexity

O(1)

The complexity of ch, lit(ch), ns::char_, ns::char_(ch), and ns::char_("c") is constant as all generators emit exactly one character per invocation.
The character range generator (\texttt{ns::char\_}(ch1, ch2)) additionally requires constant lookup time for the verification whether the attribute belongs to the character range.

The character set generator (\texttt{ns::char\_}(cs)) additionally requires \(O(\log N)\) lookup time for the verification whether the attribute belongs to the character set, where \(N\) is the number of characters in the character set.

\textbf{Example}

\begin{quote}
\textbf{Note}

The test harness for the example(s) below is presented in the Basics Examples section.
\end{quote}

Some includes:

\begin{verbatim}
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
\end{verbatim}

Some using declarations:

\begin{verbatim}
using boost::spirit::karma::lit;
using boost::spirit::ascii::char_;\end{verbatim}

Basic usage of \texttt{char\_} generators:

\begin{verbatim}
test_generator("A", 'A');
test_generator("A", lit('A'));

test_generator_attr("a", char\_, 'a');
test_generator("A", char\_('A'));
test_generator_attr("A", char\_('A'), 'A');
test_generator_attr("", char\_('A'), 'B');  // fails (as 'A' != 'B')

test_generator_attr("A", char\_('A'), 'Z');
test_generator_attr("", char\_('A'), 'Z');  // fails (as 'a' does not belong to 'A'...'Z')

test_generator_attr("k", char\_("a-z0-9"), 'k');
test_generator_attr("", char\_("a-z0-9"), 'A');  // fails (as 'A' does not belong to "a-z0-9")\end{verbatim}

\textbf{Character Classification Generators \texttt{(alnum, digit, etc.)}}

\textbf{Description}

The library has the full repertoire of single character generators for character classification. This includes the usual \texttt{alnum}, \texttt{alpha}, \texttt{digit}, \texttt{xdigit}, etc. generators. These generators have an associated Character Encoding Namespace. This is needed when doing basic operations such as forcing lower or upper case.
Header

```cpp
// forwards to <boost/spirit/home/karma/char/char_class.hpp>
#include <boost/spirit/include/karma_char_class.hpp>
```

Also, see Include Structure.

### Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns::alnum</td>
</tr>
<tr>
<td>ns::alpha</td>
</tr>
<tr>
<td>ns::blank</td>
</tr>
<tr>
<td>ns::cntrl</td>
</tr>
<tr>
<td>ns::digit</td>
</tr>
<tr>
<td>ns::graph</td>
</tr>
<tr>
<td>ns::lower</td>
</tr>
<tr>
<td>ns::print</td>
</tr>
<tr>
<td>ns::punct</td>
</tr>
<tr>
<td>ns::space</td>
</tr>
<tr>
<td>ns::upper</td>
</tr>
<tr>
<td>ns::xdigit</td>
</tr>
</tbody>
</table>

In the table above, `ns` represents a **Character Encoding Namespace** used by the corresponding character class generator. All listed generators have a mandatory attribute `Ch` and will not compile if no attribute is associated.

#### Model of

**PrimitiveGenerator**

#### Notation

`ns` A **Character Encoding Namespace**.

#### Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in **PrimitiveGenerator**.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns::alnum</td>
<td>If the mandatory attribute satisfies the concept of <code>std::isalnum</code> in the Character Encoding Namespace the generator succeeds after emitting its attribute (unless the underlying output stream reports an error). This generator fails otherwise while not generating anything.</td>
</tr>
<tr>
<td>ns::alpha</td>
<td>If the mandatory attribute satisfies the concept of <code>std::isalpha</code> in the Character Encoding Namespace the generator succeeds after emitting its attribute (unless the underlying output stream reports an error). This generator fails otherwise while not generating anything.</td>
</tr>
<tr>
<td>ns::blank</td>
<td>If the mandatory attribute satisfies the concept of <code>std::isblank</code> in the Character Encoding Namespace the generator succeeds after emitting its attribute (unless the underlying output stream reports an error). This generator fails otherwise while not generating anything.</td>
</tr>
<tr>
<td>ns::cntrl</td>
<td>If the mandatory attribute satisfies the concept of <code>std::iscntrl</code> in the Character Encoding Namespace the generator succeeds after emitting its attribute (unless the underlying output stream reports an error). This generator fails otherwise while not generating anything.</td>
</tr>
<tr>
<td>ns::digit</td>
<td>If the mandatory attribute satisfies the concept of <code>std::isdigit</code> in the Character Encoding Namespace the generator succeeds after emitting its attribute (unless the underlying output stream reports an error). This generator fails otherwise while not generating anything.</td>
</tr>
<tr>
<td>ns::graph</td>
<td>If the mandatory attribute satisfies the concept of <code>std::isgraph</code> in the Character Encoding Namespace the generator succeeds after emitting its attribute (unless the underlying output stream reports an error). This generator fails otherwise while not generating anything.</td>
</tr>
<tr>
<td>ns::print</td>
<td>If the mandatory attribute satisfies the concept of <code>std::isprint</code> in the Character Encoding Namespace the generator succeeds after emitting its attribute (unless the underlying output stream reports an error). This generator fails otherwise while not generating anything.</td>
</tr>
<tr>
<td>ns::punct</td>
<td>If the mandatory attribute satisfies the concept of <code>std::ispunct</code> in the Character Encoding Namespace the generator succeeds after emitting its attribute (unless the underlying output stream reports an error). This generator fails otherwise while not generating anything.</td>
</tr>
<tr>
<td>ns::xdigit</td>
<td>If the mandatory attribute satisfies the concept of <code>std::isxdigit</code> in the Character Encoding Namespace the generator succeeds after emitting its attribute (unless the underlying output stream reports an error). This generator fails otherwise while not generating anything.</td>
</tr>
<tr>
<td>Expression</td>
<td>Semantics</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
</tr>
<tr>
<td>ns::lower</td>
<td>If the mandatory attribute satisfies the concept of std::islower in the Character Encoding Namespace the generator succeeds after emitting its attribute (unless the underlying output stream reports an error). This generator fails otherwise while not generating anything.</td>
</tr>
<tr>
<td>ns::upper</td>
<td>If the mandatory attribute satisfies the concept of std::isupper in the Character Encoding Namespace the generator succeeds after emitting its attribute (unless the underlying output stream reports an error). This generator fails otherwise while not generating anything.</td>
</tr>
<tr>
<td>ns::space</td>
<td>If the optional attribute satisfies the concept of std::isspace in the Character Encoding Namespace the generator succeeds after emitting its attribute (unless the underlying output stream reports an error). This generator fails otherwise while not generating anything. If no attribute is supplied this generator emits a single space character in the character set defined by ns.</td>
</tr>
</tbody>
</table>

Possible values for ns are described in the section Character Encoding Namespace.

**Note**

The generators alpha and alnum might seem to behave unexpected if used inside a lower[] or upper[] directive. Both directives additionally apply the semantics of std::islower or std::isupper to the respective character class. Some examples:

```
std::string s;
std::back_insert_iterator<std::string> out(s);
generate(out, lower[alpha], 'a'); // succeeds emitting 'a'
generate(out, lower[alpha], 'A'); // fails
```

The generator directive upper[] behaves correspondingly.

**Attributes**

All listed character class generators can take any attribute Ch. All character class generators (except space) require an attribute and will fail compiling otherwise.

**Note**

In addition to their usual attribute of type Ch all listed generators accept an instance of a boost::optional<Ch> as well. If the boost::optional<> is initialized (holds a value) the generators behave as if their attribute was an instance of Ch and emit the value stored in the boost::optional<>. Otherwise the generators will fail.

**Complexity**

O(1)

The complexity is constant as the generators emit not more than one character per invocation.
Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::alpha;
using boost::spirit::karma::upper;
```

Basic usage of an `alpha` generator:

```cpp
test_generator_attr("a", alpha, 'a');
test_generator_attr("A", alpha, 'A');
test_generator_attr("", alpha, '1'); // fails (as isalpha('1') is false)
test_generator_attr("A", upper[alpha], 'A');
test_generator_attr("", upper[alpha], 'a'); // fails (as isupper('a') is false)
```

Generator Directives

This module includes different generator directives. It includes alignment directives (`left_align[]`, `center[]`, and `right_align[]`), repetition (`repeat[]`), directives controlling automatic delimiting (`verbatim[]`, `no_delimit[]`, and `delimit[]`), controlling case sensitivity (`upper[]` and `lower[]`), field width (`maxwidth[]`), buffering (`buffer[]`), splitting into columns (`columns[]`) and attribute handling (`duplicate[]`, `omit[]`, and `skip[]`).

Module Header

```cpp
#include <boost/spirit/home/karma/directive.hpp>
```

Also, see Include Structure.

Alignment Generator Directives (`left_align[](center[], right_align[])`)

Description

The alignment directives allow to left align, right align or center output emitted by other generators into columns of a specified width while using an arbitrary generator to create the padding.

Header

For the `left_align[]` directive:
For the `center[]` directive:

```cpp
#include <boost/spirit/home/karma/directive/center_alignment.hpp>
```

For the `right_align[]` directive:

```cpp
#include <boost/spirit/home/karma/directive/right_alignment.hpp>
```

Also, see `Include Structure`.

**Namespace**

<table>
<thead>
<tr>
<th>Name</th>
<th></th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::left_align // alias: boost::spirit::karma::left_align</td>
<td>boost::spirit::center // alias: boost::spirit::karma::center</td>
<td></td>
</tr>
<tr>
<td>boost::spirit::right_align // alias: boost::spirit::karma::right_align</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Model of**

UnaryGenerator

**Notation**

- `a` A generator object
- `pad` A generator object, or a Lazy Argument that evaluates to a generator object
- `A, Pad` Attribute types of the generators `a` and `pad`
- `width` Numeric literal, any unsigned integer value, or a Lazy Argument that evaluates to an unsigned integer value

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in UnaryGenerator.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>left_align[a]</td>
<td>Generate a left aligned in a column of width as defined by the preprocessor constant <code>BOOST_KARMA_DEFAULT_FIELD_LENGTH</code> (default: 10), while using <code>space</code> to emit the necessary padding. This generator succeeds as long as its embedded generator a does not fail (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>left_align(width) [a]</td>
<td>Generate a left aligned in a column of the given width, while using <code>space</code> to emit the necessary padding. This generator succeeds as long as its embedded generator a does not fail (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>left_align(pad) [a]</td>
<td>Generate a left aligned in a column of width as defined by the preprocessor constant <code>BOOST_KARMA_DEFAULT_FIELD_LENGTH</code> (default: 10), while using the generator <code>pad</code> to emit the necessary padding. This generator succeeds as long as its embedded and padding generators a and pad do not fail (except if the underlying output stream reports an error).</td>
</tr>
<tr>
<td>left_align(width, pad) [a]</td>
<td>Generate a left aligned in a column of the given width, while using the generator <code>pad</code> to emit the necessary padding. This generator succeeds as long as its embedded and padding generators a and pad do not fail (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>center[a]</td>
<td>Generate a centered in a column of width as defined by the preprocessor constant <code>BOOST_KARMA_DEFAULT_FIELD_LENGTH</code> (default: 10), while using <code>space</code> to emit the necessary padding. This generator succeeds as long as its embedded generator a does not fail (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>center(width) [a]</td>
<td>Generate a centered in a column of the given width, while using <code>space</code> to emit the necessary padding. This generator succeeds as long as its embedded generator a does not fail (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>center(pad) [a]</td>
<td>Generate a centered in a column of width as defined by the preprocessor constant <code>BOOST_KARMA_DEFAULT_FIELD_LENGTH</code> (default: 10), while using the generator <code>pad</code> to emit the necessary padding. This generator succeeds as long as its embedded and padding generators a and pad do not fail (except if the underlying output stream reports an error).</td>
</tr>
<tr>
<td>center(width, pad) [a]</td>
<td>Generate a centered in a column of the given width, while using the generator <code>pad</code> to emit the necessary padding. This generator succeeds as long as its embedded and padding generators a and pad do not fail (unless the underlying output stream reports an error).</td>
</tr>
</tbody>
</table>
### Expression | Semantics
--- | ---
right_align[a] | Generate a right aligned in a column of width as defined by the preprocessor constant BOOST_KARMA_DEFAULT_FIELD_LENGTH (default: 10), while using `space` to emit the necessary padding. This generator succeeds as long as its embedded generator `a` does not fail (unless the underlying output stream reports an error).

right_align(width)[a] | Generate a right aligned in a column of the given `width`, while using `space` to emit the necessary padding. This generator succeeds as long as its embedded generator `a` does not fail (unless the underlying output stream reports an error).

right_align(pad)[a] | Generate a right aligned in a column of width as defined by the preprocessor constant BOOST_KARMA_DEFAULT_FIELD_LENGTH (default: 10), while using the generator `pad` to emit the necessary padding. This generator succeeds as long as its embedded and padding generators `a` and `pad` do not fail (except if the underlying output stream reports an error).

right_align(width, pad)[a] | Generate a right aligned in a column of the given `width`, while using the generator `pad` to emit the necessary padding. This generator succeeds as long as its embedded and padding generators `a` and `pad` do not fail (unless the underlying output stream reports an error).

---

**Note**

None of the generator directives listed above limits the emitted output to the respective column width. If the emitted output is longer than the specified (or implied) column width, the generated output overruns the column to the right.

If the output needs to be limited to a specified column width, use the `maxwidth[]` directive, for instance:

```
maxwidth(8)[right_align(12)["1234567890"]]
```

which will output (without the quotes):

```
" 123456"
```

**Attributes**

See Compound Attribute Notation.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>left_align[]</td>
<td>a: A --&gt; left_align[a]: A</td>
</tr>
<tr>
<td>left_align(width)[]</td>
<td>a: A --&gt; left_align(width)[a]: A</td>
</tr>
<tr>
<td>left_align(pad)[]</td>
<td>a: A, pad: Pad --&gt; left_align(pad)[a]: A</td>
</tr>
<tr>
<td>left_align(pad, width)[]</td>
<td>a: A, pad: Pad --&gt; left_align(pad, width)[a]: A</td>
</tr>
<tr>
<td>center[]</td>
<td>a: A --&gt; center[a]: A</td>
</tr>
<tr>
<td>center(width)[]</td>
<td>a: A --&gt; center(width)[a]: A</td>
</tr>
<tr>
<td>center(pad)[]</td>
<td>a: A, pad: Pad --&gt; center(pad)[a]: A</td>
</tr>
<tr>
<td>center(pad, width)[]</td>
<td>a: A, pad: Pad --&gt; center(pad, width)[a]: A</td>
</tr>
<tr>
<td>right_align[]</td>
<td>a: A --&gt; right_align[a]: A</td>
</tr>
<tr>
<td>right_align(width)[]</td>
<td>a: A --&gt; right_align(width)[a]: A</td>
</tr>
<tr>
<td>right_align(pad)[]</td>
<td>a: A, pad: Pad --&gt; right_align(pad)[a]: A</td>
</tr>
</tbody>
</table>
Complexity

The overall complexity of an alignment generator directive is defined by the complexity of its embedded and padding generator. The complexity of the left alignment directive generator itself is O(1). The complexity of the center and right alignment directive generators is O(N), where N is the number of characters emitted by the embedded and padding generators.

Example

The test harness for the example(s) below is presented in the Basics Examples section.

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>

using boost::spirit::karma::double_;  
using boost::spirit::karma::left_align;
using boost::spirit::karma::center;
using boost::spirit::karma::right_align;

std::pair<double, double> p (1.0, 2.0);
test_generator_attr("1.0[2.0]", left_align(8)[double_] << '|' << double_, p);
test_generator_attr("1.0[2.0]", center(8)[double_] << '|' << double_, p);
test_generator_attr("1.0[2.0]", right_align(8)[double_] << '|' << double_, p);
```

Repetition Generator Directive (repeat[1])

Description

The repetition directive allows to repeat an arbitrary generator expression while optionally specifying the lower and upper repetition counts. It provides a more powerful and flexible mechanism for repeating a generator. There are grammars that are impractical and cumbersome, if not impossible, for the basic EBNF iteration syntax (unary '*') and the unary '+') to specify. Examples:

- A file name may have a maximum of 255 characters only.
- A specific bitmap file format has exactly 4096 RGB color information.
A 256 bit binary string (1..256 1s or 0s).

### Header

```cpp
// forwards to <boost/spirit/home/karma/directive/repeat.hpp>
#include <boost/spirit/include/karma_repeat.hpp>
```

Also, see Include Structure.

### Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::repeat // alias: boost::spirit::karma::repeat</td>
</tr>
<tr>
<td>boost::spirit::inf // alias: boost::spirit::karma::inf</td>
</tr>
</tbody>
</table>

### Model of

UnaryGenerator

### Notation

- `a` A generator object
- `num, num1, num2` Numeric literals, any unsigned integer value, or a Lazy Argument that evaluates to an unsigned integer value
- `inf` Placeholder expression standing for 'no upper repeat limit'

### Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryGenerator.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeat[a]</td>
<td>Repeat the generator a zero or more times. This generator succeeds as long as its embedded generator a does not fail (except if the underlying output stream reports an error). This variant of repeat[] is semantically equivalent to the Kleene Star operator *a.</td>
</tr>
<tr>
<td>repeat(num) [a]</td>
<td>Repeat the generator a exactly num times. This generator succeeds as long as its embedded generator a does not fail and as long as the associated attribute (container) contains at least num elements (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>repeat(num1, num2) [a]</td>
<td>Repeat the generator a at least num1 times but not more than num2 times. This generator succeeds as long as its embedded generator a does not fail and as long as the associated attribute (container) contains at least num1 elements (unless the underlying output stream reports an error). If the associated attribute (container) does contain more than num2 elements, this directive limits the repeat count to num2.</td>
</tr>
<tr>
<td>repeat(num, inf) [a]</td>
<td>Repeat the generator a at least num1 times. No upper limit for the repeat count is set. This generator succeeds as long as its embedded generator a does not fail and as long as the associated attribute (container) contains at least num1 elements (unless the underlying output stream reports an error).</td>
</tr>
</tbody>
</table>

**Note**

All failing iterations of the embedded generator will consume one element from the supplied attribute. The overall repeat[a] will succeed as long as the iteration criteria (number of successful invocations of the embedded generator) is fulfilled (unless the underlying output stream reports an error).

**Attributes**

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeat[a]</td>
<td>a: A --&gt; repeat[a]: vector&lt;A&gt;</td>
</tr>
<tr>
<td></td>
<td>a: Unused --&gt; repeat[a]: Unused</td>
</tr>
<tr>
<td>repeat(num) [a]</td>
<td>a: A --&gt; repeat(num) [a]: vector&lt;A&gt;</td>
</tr>
<tr>
<td></td>
<td>a: Unused --&gt; repeat(num) [a]: Unused</td>
</tr>
<tr>
<td>repeat(num1, num2) [a]</td>
<td>a: A --&gt; repeat(num1, num2) [a]: vector&lt;A&gt;</td>
</tr>
<tr>
<td></td>
<td>a: Unused --&gt; repeat(num1, num2) [a]: Unused</td>
</tr>
<tr>
<td>repeat(num, inf) [a]</td>
<td>a: A --&gt; repeat(num, inf) [a]: vector&lt;A&gt;</td>
</tr>
<tr>
<td></td>
<td>a: Unused --&gt; repeat(num, inf) [a]: Unused</td>
</tr>
</tbody>
</table>
Important

The table above uses vector\(<A>\) as placeholders only.

The notation of vector\(<A>\) stands for any STL container holding elements of type \(A\).

It is important to note, that the \texttt{repeat[\]} directive does not perform any buffering of the output generated by its embedded elements. That means that any failing element generator might have already generated some output, which is \textit{not} rolled back.

Tip

The simplest way to force a \texttt{repeat[\]} directive to behave as if it did buffering is to wrap it into a buffering directive (see \texttt{buffer}):

\begin{verbatim}
buffer[repeat[a]]
\end{verbatim}

which will \textit{not} generate any output in case of a failing generator \texttt{repeat[a]}. The expression:

\begin{verbatim}
repeat[buffer[a]]
\end{verbatim}

will not generate any partial output from a generator \(a\) if it fails generating in the middle of its output. The overall expression will still generate the output as produced by all succeeded invocations of the generator \(a\).

Complexity

The overall complexity of the repetition generator is defined by the complexity of its embedded generator. The complexity of the repeat itself is \(O(N)\), where \(N\) is the number of repetitions to execute.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

\begin{verbatim}
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
\end{verbatim}

Some using declarations:

\begin{verbatim}
using boost::spirit::karma::double_;
using boost::spirit::karma::repeat;
\end{verbatim}

Basic usage of \texttt{repeat} generator directive:
std::vector<double> v;
   v.push_back(1.0);
   v.push_back(2.0);
   v.push_back(3.0);

test_generator_attr("[1.0][2.0][3.0]", repeat(2)["[ " << double_ << "]"], v);
// fails because of insufficient number of items
    test_generator_attr("", repeat(4)["[ " << double_ << "]"], v);

Generator Directives Controlling Automatic Delimiting (verbatim[], no_delimit[], delimit[])  

Description

The directives delimit[], no_delimit[], and verbatim[] can be used to control automatic delimiting. The directives verbatim[] and no_delimit[] disable any automatic delimiting, while the directive delimit[] (re-)enables automatic delimiting.

Header

For the verbatim[] directive:

    // forwards to <boost/spirit/home/karma/directive/verbatim.hpp>
    #include <boost/spirit/include/karma_verbatim.hpp>

For the no_delimit[] directive:

    // forwards to <boost/spirit/home/karma/directive/no_delimit.hpp>
    #include <boost/spirit/include/karma_no_delimit.hpp>

For the delimit[] directive:

    // forwards to <boost/spirit/home/karma/directive/delimit.hpp>
    #include <boost/spirit/include/karma_delimit.hpp>

Also, see Include Structure.

Namespace

Name

boost::spirit::verbatim // alias: boost::spirit::karma::verbatim
boost::spirit::no_delimit // alias: boost::spirit::karma::no_delimit
boost::spirit::delimit // alias: boost::spirit::karma::delimit

Model of

UnaryGenerator

Notation

a  A generator object

d  A generator object, or a Lazy Argument that evaluates to a generator object
Attribute types of the generators a and d

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryGenerator.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>delimit[a]</td>
<td>Enable automatic delimiting for the embedded generator a while using the space generator as the delimiting generator. If used inside a verbatim[] directive it re-enables the delimiter generator as used outside of this verbatim[] instead. The directive succeeds as long as the embedded generator succeeded (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>delimit (d) [a]</td>
<td>Enable automatic delimiting for the embedded generator a while using the generator d as the delimiting generator. The directive succeeds as long as the embedded generator succeeded (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>verbatim[a]</td>
<td>Disable automatic delimiting for the embedded generator a. The directive succeeds as long as the embedded generator succeeded (unless the underlying output stream reports an error). This directive it has no effect if it is used when no delimiting is active. When delimiting is active this directive performs a post-delimit step (which is different from the behavior of no_delimit[]).</td>
</tr>
<tr>
<td>no_delimit[a]</td>
<td>Disable automatic delimiting for the embedded generator a. The directive succeeds as long as the embedded generator succeeded (unless the underlying output stream reports an error). This directive it has no effect if it is used when no delimiting is active. When delimiting is active this directive does not perform a post-delimit step (which is different from the behavior of verbatim[]).</td>
</tr>
</tbody>
</table>

Attributes

See Compound Attribute Notation.
### Expression

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Expression</th>
</tr>
</thead>
</table>
| delimit[a] | a: A --> delimit[a]: A  
|           | a: Unused --> delimit[a]: Unused |
| delimit(d)[a] | a: A, d: D --> delimit(d)[a]: A  
|               | a: Unused, d: D --> delimit(d)[a]: Unused |
| verbatim[a] | a: A --> verbatim[a]: A  
|            | a: Unused --> verbatim[a]: Unused |
| no_delimit[a] | a: A --> no_delimit[a]: A  
|               | a: Unused --> no_delimit[a]: Unused |

### Complexity

The overall complexity of the generator directives `delimit[]`, `verbatim[]`, and `no_delimit[]` is defined by the complexity of its embedded generators. The complexity of the directives themselves is O(1).

### Example

#### Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::double_;  
using boost::spirit::karma::delimit;  
using boost::spirit::karma::verbatim;  
```

Basic usage of `delimit` generator directive:
Generator Directives Controlling Case Sensitivity ($\texttt{upper[]}$, $\texttt{lower[]}$)

**Description**

The generator directives $\texttt{ns::lower[]}$ and $\texttt{ns::upper[]}$ force their embedded generators to emit lower case or upper case only characters based on the interpretation of the generated characters in the character set defined by $\texttt{ns}$ (see Character Encoding Namespace).

**Header**

```cpp
// forwards to <boost/spirit/home/karma/directive/upper_lower_case.hpp>
#include <boost/spirit/include/karma_upper_lower_case.hpp>
```

Also, see Include Structure.

**Namespace**

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\texttt{ns::lower}$</td>
</tr>
<tr>
<td>$\texttt{ns::upper}$</td>
</tr>
</tbody>
</table>

In the table above, $\texttt{ns}$ represents a Character Encoding Namespace.

**Model of**

The model of $\texttt{lower[]}$ and $\texttt{upper[]}$ is the model of its subject generator.

**Notation**

- $a$ A generator object
- $A$ Attribute type of the generator $a$
- $\texttt{ns}$ A Character Encoding Namespace.

**Expression Semantics**

The $\texttt{lower[]}$ and $\texttt{upper[]}$ directives have no special generator semantics. They are pure modifier directives. They indirectly influence the way all subject generators work. They add information (the $\texttt{tag::upper}$ or $\texttt{tag::lower}$) to the Modifier template parameter used while transforming the $\texttt{proto::expr}$ into the corresponding generator expression. This is achieved by the following specializations:
namespace boost { namespace spirit {

    template <typename CharEncoding>
    struct is_modifier_directive<
        karma::domain,
        tag::char_code<tag::lower, CharEncoding>
    > : mpl::true_ {
    }

    template <typename CharEncoding>
    struct is_modifier_directive<
        karma::domain,
        tag::char_code<tag::upper, CharEncoding>
    > : mpl::true_ {
    }
}

(for more details see the section describing the compilation process of the Boost.Proto expression into the corresponding generator expressions).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns::lower[a]</td>
<td>Generate a as lower case, interpreted in the character set defined by ns. The directive succeeds as long as the embedded generator succeeded (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>ns::upper[a]</td>
<td>Generate a as upper case, interpreted in the character set defined by ns. The directive succeeds as long as the embedded generator succeeded (unless the underlying output stream reports an error).</td>
</tr>
</tbody>
</table>

**Note**

If both directives are 'active' with regard to a generator, the innermost of those directives takes precedence. For instance:

```
generate(sink, ascii::lower['A' << ascii::upper['b']])
```

will generate "aB" (without the quotes).

Further, the directives will have no effect on generators emitting characters not having an upper case or lower case equivalent in the character set defined by ns.

**Attributes**

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
</table>
| ns:lower[a] | a: A --> ns:lower[a]: A  
 a: Unused --> ns:lower[a]: Unused |
| ns:upper[a] | a: A --> ns:upper[a]: A  
 a: Unused --> ns:upper[a]: Unused |
**Complexity**

The overall complexity of the generator directives `ns::lower[]` and `ns::upper[]` is defined by the complexity of its embedded generators. The directives themselves are compile time only directives, having no impact on runtime performance.

**Example**

```cpp
#include <boost/spirit/include/karma.hpp>
#include <iostream>
#include <string>

using boost::spirit::karma::double_
using boost::spirit::ascii::upper;
using boost::spirit::ascii::lower;

test_generator_attr("abc:2.0e-06", lower["ABC:" << double_], 2e-6);
test_generator_attr("ABC:2.0E-06", upper["abc:" << double_], 2e-6);
```

**Generator Directives Controlling the Maximum Field Width (`maxwidth[]`)**

**Description**

The `maxwidth[]` directive allows to limit (truncate) the overall length of the output generated by the embedded generator.

**Header**

```cpp
// forwards to <boost/spirit/home/karma/directive/maxwidth.hpp>
#include <boost/spirit/include/karma_maxwidth.hpp>
```

Also, see [Include Structure](#).

**Name**

`boost::spirit::maxwidth` // alias: `boost::spirit::karma::maxwidth`

**Model of**

`UnaryGenerator`

**Notation**

<table>
<thead>
<tr>
<th>a</th>
<th>A generator object</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Attribute type of the generator a</td>
</tr>
</tbody>
</table>
num Numeric literal, any unsigned integer value, or a Lazy Argument that evaluates to an unsigned integer value

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryGenerator.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxwidth[a]</td>
<td>Limit the overall length of the emitted output of the embedded generator (including characters generated by automatic delimiting) to the number of characters as defined by the preprocessor constant BOOST_KARMA_DEFAULT_FIELD_MAXWIDTH. Any additional output is truncated. The directive succeeds as long as the embedded generator succeeded (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>maxwidth(num)[a]</td>
<td>Limit the overall length of the emitted output of the embedded generator (including characters generated by automatic delimiting) to the number of characters as defined by num. Any additional output is truncated. The directive succeeds as long as the embedded generator succeeded (unless the underlying output stream reports an error).</td>
</tr>
</tbody>
</table>

Note

The maxwidth[] generator directive does not pad the generated output to fill the specified column width. If the emitted output is shorter than the specified (or implied) column width, the generated output will be more narrow than the column width.

If the output needs to always be equal to a specified column width, use one of the alignment directives left-align[], center[], or right_align[], for instance:

```
maxwidth(8)[left_align(8)"1234"]
```

which will output: "1234  " (without the quotes).

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxwidth[a]</td>
<td>a: A -&gt; maxwidth[a]: A</td>
</tr>
<tr>
<td></td>
<td>a: Unused -&gt; maxwidth[a]: Unused</td>
</tr>
<tr>
<td>maxwidth(num)[a]</td>
<td>a: A -&gt; maxwidth(num)[a]: A</td>
</tr>
<tr>
<td></td>
<td>a: Unused -&gt; maxwidth(num)[a]: Unused</td>
</tr>
</tbody>
</table>

Complexity

The overall complexity of the generator directive maxwidth[] is defined by the complexity of its embedded generator. The complexity of the directive itself is O(N), where N is the number of characters generated by the maxwidth directive.
Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::double_;  
using boost::spirit::karma::maxwidth; 
using boost::spirit::karma::left_align;  
using boost::spirit::karma::right_align; 
```

Basic usage of maxwidth generator directive:

```cpp
test_generator("01234", maxwidth(5)["0123456789"]); 
test_generator("  012", maxwidth(5)[right_align(12)["0123456789"]]); 
test_generator("0123    ", maxwidth(8)[left_align(8)["0123"]]); 
```

Generator Directive for Temporary Output Buffering (`buffer[]`)

Description

All generator components (except the Alternative (`|`) generator) pass their generated output directly to the underlying output stream. If a generator fails halfway through, the output generated so far is not ‘rolled back’. The buffering generator directive allows to avoid this unwanted output to be generated. It temporarily redirects the output produced by the embedded generator into a buffer. This buffer is flushed to the underlying stream only after the embedded generator succeeded, but is discarded otherwise.

Header

```cpp
// forwards to <boost/spirit/home/karma/directive/buffer.hpp>
#include <boost/spirit/include/karma_buffer.hpp>
```

Also, see Include Structure.

Name

```cpp
boost::spirit::buffer // alias: boost::spirit::karma::buffer
```

Model of

UnaryGenerator

Notation

a  A generator object

A  Attribute type of generator a
Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryGenerator.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer[a]</td>
<td>The embedded generator a is invoked but its output is temporarily intercepted and stored in an internal buffer. If a succeeds the buffer content is flushed to the underlying output stream, otherwise the buffer content is discarded. The buffer directive succeeds as long as the embedded generator succeeded (unless the underlying output stream reports an error).</td>
</tr>
</tbody>
</table>

**Tip**

If you want to make the buffered generator succeed regardless of the outcome of the embedded generator, simply wrap the `buffer[a]` into an additional optional: `-buffer[a]` (see Optional (unary -)).

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
</table>
| buffer[a]        | a: A --> buffer[a]: A  
|                  | a: Unused --> buffer[a]: Unused                                           |

Complexity

The overall complexity of the buffering generator directive is defined by the complexity of its embedded generator. The complexity of the buffering directive generator itself is O(N), where N is the number of characters buffered.

Example

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```c++
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:
using boost::spirit::karma::double_;  
using boost::spirit::karma::buffer;

Basic usage of a buffering generator directive. It shows how the partial output generated in the first example does not show up in the generated output as the plus generator fails (no data is available, see Plus (unary +)).

```cpp
std::vector<double> v;  // empty container
v.push_back(1.0);  // now, fill the container
v.push_back(2.0);

// empty container
std::vector<double> v;
test_generator_attr("", -buffer['' << +double_ << '"'], v);

// now, fill the container
v.push_back(1.0);
v.push_back(2.0);
test_generator_attr("[1.02.0]", buffer['' << +double_ << '"'], v);
```

### Generator Directives Consuming Attributes (omit[] and skip[])

#### Description

The directives `omit[]` and `skip[]` consumes the attribute type of the embedded generator without generating any output. The `omit[]` directive will still execute the embedded generator while discarding the generated output afterwards. The `skip[]` directive will not execute the embedded generator, but will use it only to extract the exposed attribute type.

#### Header

```cpp
// forwards to <boost/spirit/home/karma/directive/omit.hpp>
#include <boost/spirit/include/karma_omit.hpp>
```

Also, see Include Structure.

#### Name

- `boost::spirit::omit` // alias: `boost::spirit::karma::omit`
- `boost::spirit::skip` // alias: `boost::spirit::karma::skip`

#### Model of

- `UnaryGenerator`

#### Notation

- `a` A generator object
- `A` Attribute type of generator `a`

#### Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in `UnaryGenerator`.  

---

**Spirit 2.5**

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at [http://www.renderx.com](http://www.renderx.com/)
The `omit` directive consumes the attribute type of the embedded generator `A` without generating any output. It succeeds always. The embedded generator is executed and any generated output is discarded.

The `skip` directive consumes the attribute type of the embedded generator `A` without generating any output. It succeeds always. The embedded generator is not executed.

### Attributes

See Compound Attribute Notation.

### Complexity

The overall complexity of the `omit[]` directive depends on the complexity of the embedded generator. The overall complexity of the `skip[]` generator directive is O(1) as it does not generate any output.

### Example

#### Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::double_;
using boost::spirit::karma::omit;
```

Basic usage of a `omit` generator directive. It shows how it consumes the first element of the provided attribute without generating anything, leaving the second element of the attribute to the non-wrapped `double_` generator.
```cpp
std::pair<double, double> p (1.0, 2.0);
test_generator_attr("2.0", omit[double_] << double_, p);
```

Generally, this directive is helpful in situations, where the attribute type contains more information (elements) than need to be used to generate the required output. Normally in such situations we would resolve to use semantic actions to explicitly pass the correct parts of the overall attribute to the generators. The `omit` directive helps achieving the same without having to use semantic actions.

Consider the attribute type:

```cpp
typedef fusion::vector<int, double, std::string> attribute_type;
```

where we need to generate output only from the first and last element:

```cpp
typedef std::back_insert_iterator<std::string> iterator_type;

karma::rule<iterator_type, attribute_type()> r;
r = int_[_1 = phoenix::at_c<0>(_val)] << string[_1 = phoenix::at_c<2>(_val)];

std::string str;
iterator_type sink(str);
generate(sink, r, attribute_type(1, 2.0, "example")); // will generate: '1example'
```

This is error prone and not really readable. The same can be achieved by using the `omit` directive:

```cpp
r = int_ << omit[double_] << string;
```

which is at the same time more readable and more efficient as we don't have to use semantic actions.

The semantics of using the `skip[]` directive are identical to the `omit[]` directive, except that it does not actually execute the embedded generator. For this reason it is usually preferable to utilize the `skip[]` directive instead of the `omit[]` directive. On the other hand, the `omit[]` directive is very useful whenever the embedded generator produces side effects (has semantic actions which need to be executed).

**Generator Directive Duplicating Attributes (duplicate[])**

**Description**

The directive `duplicate[]` duplicates its attribute to all elements of the embedded generator if this is a sequence generator. Otherwise it does nothing.

**Header**

```cpp
#include <boost/spirit/include/karma_duplicate.hpp>
```

Also, see Include Structure.

**Name**

```cpp
boost::spirit::duplicate // alias: boost::spirit::karma::duplicate
```

**Model of**

```cpp
UnaryGenerator
```
**Notation**

- A generator object
- A Attribute type of generator a

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in `UnaryGenerator`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>duplicate[a]</td>
<td>The <code>duplicate</code> directive duplicates the supplied attribute for all elements of an embedded sequence generator. For all other types of embedded generators it has no effect. It succeeds as long as its embedded generator does not fail.</td>
</tr>
</tbody>
</table>

**Attributes**

See [Compound Attribute Notation](#).  

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
</table>
| duplicate[a] | a: A --> duplicate[a]: A  
|              | a: tuple<A, A, ...> --> duplicate[a]: A  
|              | a: Unused --> duplicate[a]: Unused                                                                                                                                                                         |

If the embedded generator of the `duplicate[]` directive is a sequence it is expected that all elements of this sequence expose either the same attribute type, a compatible attribute type, or unused. In this case, the `duplicate[]` directive exposes the attribute type of its first element. The behavior of the `duplicate[]` directive is undefined if the elements of an embedded sequence do not expose the same attributes. Most likely, the corresponding expression will not compile.

**Complexity**

The overall complexity of the `duplicate[]` directive depends on the complexity of the embedded generator.

**Example**

**Note**

The test harness for the example(s) below is presented in the [Basics Examples](#) section.

Some includes:

```
#include <boost/spirit/include/karma.hpp>  
#include <boost/spirit/include/support_utree.hpp>  
#include <boost/spirit/include/phoenix_core.hpp>  
#include <boost/spirit/include/phoenix_operator.hpp>  
#include <boost/fusion/include/std_pair.hpp>  
#include <iostream>  
#include <string> 
```

Some using declarations:
Basic usage of the \texttt{duplicate} generators:

\begin{verbatim}
using boost::spirit::karma::double_;  
using boost::spirit::karma::duplicate;  
using boost::spirit::karma::space;  

Basic usage of the duplicate generators:

\begin{verbatim}
test_generator_attr("2.02.0", duplicate[double_ << double_], 2.0);  
test_generator_attr_delim("2.0 2.0 ", duplicate[double_ << double_], space, 2.0);  
\end{verbatim}

\textbf{Generator Directive Separating Output Into Columns (columns[])}

\textbf{Description}

The \texttt{columns[]} directive separates the output emitted by the embedded generator by inserting special column separators.

\textbf{Header}

\begin{verbatim}
// forwards to \texttt{<boost/spirit/home/karma/directive/columns.hpp>}
#include \texttt{<boost/spirit/include/karma_columns.hpp>}
\end{verbatim}

Also, see \texttt{Include Structure}.

\textbf{Name}

\begin{verbatim}
boost::spirit::columns // alias: boost::spirit::karma::columns
\end{verbatim}

\textbf{Model of}

\texttt{UnaryGenerator}

\textbf{Notation}

\begin{itemize}
  \item \texttt{a} A generator object
  \item \texttt{g} A generator object, or a \texttt{Lazy Argument} that evaluates to a generator object, will be used to emit column separators
  \item \texttt{A} Attribute type of generator \texttt{a}
  \begin{itemize}
    \item \texttt{[num]} Numeric literal, any unsigned integer value, or a \texttt{Lazy Argument} that evaluates to an unsigned integer value defining the number of items to emit in between the column separators
  \end{itemize}
\end{itemize}

\textbf{Expression Semantics}

Semantics of an expression is defined only where it differs from, or is not defined in \texttt{UnaryGenerator}.  

---
The columns directive invokes a generator after each N-th element of the embedded generator has been emitted. The number of columns is defined by the preprocessor constant `BOOST_KARMA_DEFAULT_COLUMNS`. The column separator used will be `karma::eol`.

The columns directive invokes a generator after each N-th element of the embedded generator has been emitted. The number of columns is defined by the argument to the directive `num`. The column separator used will be `karma::eol`.

The columns directive invokes a generator after each N-th element of the embedded generator has been emitted. The number of columns is defined by the preprocessor constant `BOOST_KARMA_DEFAULT_COLUMNS`. The column separator used will be `g`.

The columns directive invokes a generator after each N-th element of the embedded generator has been emitted. The number of columns is defined by the argument to the directive `num`. The column separator used will be `g`.

**Attributes**

See Compound Attribute Notation.

**Complexity**

The overall complexity of the `columns` generator directive depends on the complexity of the embedded generator. The complexity of the `columns` generator directive itself is O(N), where N is the number of inserted column separators.
Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::double_; 
using boost::spirit::karma::columns;
using boost::spirit::karma::space;
```

Basic usage of the `columns` generators:

```cpp
std::vector<double> v;
v.push_back(1.0);
v.push_back(2.0);
v.push_back(3.0);
test_generator_attr("1.0\n2.0\n3.0\n", columns(1)[*double_], v);
test_generator_attr_delim("1.0 2.0 \
3.0 \
", columns(2)[*double_], space, v);
```

Generator Directives Forcing Atomic Extraction (`as<T>`, `as_string[]`, `as_wstring[]`)

Description

The `as<T>` class forces the atomic extraction of a container type `T` from it’s consumed attribute. Usually, repetitive generators (such as Kleene `*`, etc) or sequences exposing a `vector<A>` will extract elements from the container supplied as their consumed attribute by looping through the containers iterators. In some cases, this may be undesirable. The `as<T>` class creates a directive that will pass an unnamed temporary object of type `T` to it’s subject, if extracting `T` from it’s consumed attribute determined at generation-time to be valid. `traits::valid_as<T>()` is called by `as<T>` to determine validity; if it returns false, the generator fails. Subsequent extraction is performed by calling `traits::as<T>()`.

Note

`T` is required to be a container type. If `traits::is_container` does not return true for `T`, a compile-time error will occur.

Header

```cpp
// forwards to <boost/spirit/home/karma/directive/as.hpp>
#include <boost/spirit/include/karma_as.hpp>
```

Also, see Include Structure.
Namespace

Name
boost::spirit::as_string // alias: boost::spirit::karma::as_string
boost::spirit::as_wstring // alias: boost::spirit::karma::as_wstring

Synopsis

```cpp
template <typename T>
struct as;
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>A container type.</td>
<td>none</td>
</tr>
</tbody>
</table>

Model of

UnaryGenerator

Notation

```
a          A Generator.
attr       The attribute supplied to the directive.
```

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryGenerator.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>as&lt;T&gt;()[a]</code></td>
<td>Extract an instance of T from attr, and invoke the subject generator a, supplying the unnamed temporary as it's attribute.</td>
</tr>
<tr>
<td><code>as_string[a]</code></td>
<td>Equivalent to <code>as&lt;std::string&gt;()[a]</code></td>
</tr>
<tr>
<td><code>as_wstring[a]</code></td>
<td>Equivalent to <code>as&lt;std::wstring&gt;()[a]</code></td>
</tr>
</tbody>
</table>

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>as&lt;T&gt;()[a]</code></td>
<td><code>a: A -&gt; as&lt;T&gt;()[a]: T</code></td>
</tr>
</tbody>
</table>

Complexity

The complexity is defined by the complexity of the subject generator, a, and the complexity of the extraction unnamed container of type T from the attribute attr.
Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some using declarations:

```cpp
using boost::spirit::utree;
using boost::spirit::utree_type;
using boost::spirit::utf8_symbol_type;
using boost::spirit::karma::as;
using boost::spirit::karma::as_string;
using boost::spirit::karma::char_;
using boost::spirit::karma::double_;
```

Simple usage of as<T>, as_string and as_wstring:

To properly handle string concatenation with utree, we make use of as_string[]. We also use as<T> to explicitly extract a utree symbol node.

```cpp
typedef as<utf8_symbol_type> as_symbol_type;
as_symbol_type const as_symbol = as_symbol_type();

utree ut;
ut.push_back("xyz");
ut.push_back(1.23);

test_generator_attr("xyz1.23", as_string[*char_] << double_, ut);
test_generator_attr("xyz1.23", as<std::string>()[*char_] << double_, ut);

ut.clear();
ut.push_back(utf8_symbol_type("xyz"));
ut.push_back(1.23);

test_generator_attr("xyz1.23", as_symbol[*char_] << double_, ut);
test_generator_attr("xyz1.23", as<utf8_symbol_type>()[*char_] << double_, ut);
```

Nonterminal Generators

Module Headers

```cpp
// forwards to <boost/spirit/home/karma/nonterminal.hpp>
#include <boost/spirit/include/karma_nonterminal.hpp>
```

Also, see Include Structure.

Generator Rule

Description

The rule is a polymorphic generator that acts as a named place-holder capturing the behavior of a PEG expression assigned to it. Naming a Parsing Expression Grammar expression allows it to be referenced later and makes it possible for the rule to call itself. This is one of the most important mechanisms and the reason behind the word “recursive” in recursive descent output generation.
Header

```cpp
// forwards to <boost/spirit/home/karma/nonterminal/rule.hpp>
#include <boost/spirit/include/karma_rule.hpp>
```

Also, see Include Structure.

Namespace

Name

`boost::spirit::karma::rule`

Synopsis

```cpp
template <typename OutputIterator, typename A1, typename A2, typename A3>
struct rule;
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputIterator</td>
<td>The underlying output iterator type that the rule is expected to work on.</td>
<td>none</td>
</tr>
<tr>
<td>A1, A2, A3</td>
<td>Either Signature, Delimiter or Locals in any order. See table below.</td>
<td>See table below.</td>
</tr>
</tbody>
</table>

Here is more information about the template parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>Specifies the rule's consumed (value to output) and inherited (arguments) attributes. More on this here: Nonterminal.</td>
<td>unused_type. When Signature defaults to unused_type, the effect is the same as specifying a signature of void(), which is also equivalent to unused_type()</td>
</tr>
<tr>
<td>Delimiter</td>
<td>Specifies the rule's delimiter generator. Specify this if you want the rule to delimit the generated output.</td>
<td>unused_type</td>
</tr>
<tr>
<td>Locals</td>
<td>Specifies the rule's local variables. See Nonterminal.</td>
<td>unused_type</td>
</tr>
</tbody>
</table>

Model of

`Nonterminal`

Notation

- `r, r2` Rules
- `g` A generator expression
- `OutputIterator` The underlying output iterator type that the rule is expected to work on.
A1, A2, A3

Either Signature, Delimiter or Locals in any order.

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in **Nonterminal**.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule&lt;OutputIterator, A1, A2, A3&gt; [ r(name); ]</td>
<td>Rule declaration. <strong>OutputIterator</strong> is required. <strong>A1, A2, A3</strong> are optional and can be specified in any order. <strong>name</strong> is an optional string that gives the rule its name, useful for debugging.</td>
</tr>
<tr>
<td>rule&lt;OutputIterator, A1, A2, A3&gt; [ r(r2); ]</td>
<td>Copy construct rule <strong>r</strong> from rule <strong>r2</strong>.</td>
</tr>
<tr>
<td>[ r = r2; ]</td>
<td>Assign rule <strong>r2</strong> to <strong>r</strong>.</td>
</tr>
<tr>
<td>[ r.alias() ]</td>
<td>Return an alias of <strong>r</strong>. The alias is a generator that holds a reference to <strong>r</strong>. Reference semantics.</td>
</tr>
<tr>
<td>[ r.copy() ]</td>
<td>Get a copy of <strong>r</strong>.</td>
</tr>
<tr>
<td>[ r = g; ]</td>
<td>Rule definition</td>
</tr>
<tr>
<td>[ r %= g; ]</td>
<td>Auto-rule definition. The attribute of <strong>g</strong> should be compatible with the consumed attribute of <strong>r</strong>.</td>
</tr>
<tr>
<td>[ r.name() ]</td>
<td>Retrieve the current name of the rule object.</td>
</tr>
<tr>
<td>[ r.name(name) ]</td>
<td>Set the current name of the rule object to be <strong>name</strong>.</td>
</tr>
</tbody>
</table>

**Attributes**

The rule’s generator attribute is **RT**: The consumed attribute of the rule. See **Attribute**

**Complexity**

The complexity is defined by the complexity of the RHS generator, **g**

**Example**

**Note**

The test harness for the example(s) below is presented in the **Basics Examples** section.

Some using declarations:

```cpp
using boost::spirit::karma::rule;
using boost::spirit::karma::int_;
using boost::spirit::ascii::space;
using boost::spirit::ascii::space_type;
```

Basic rule:
Generator Grammar

Description

The grammar encapsulates a set of rules (as well as primitive generators (PrimitiveGenerator) and sub-grammars). The grammar is the main mechanism for modularization and composition. Grammars can be composed to form more complex grammars.

Header

```
// forwards to <boost/spirit/home/karma/nonterminal/grammar.hpp>
#include <boost/spirit/include/karma_grammar.hpp>
```

Also, see Include Structure.

Namespace

```
boost::spirit::karma::grammar
```

Synopsis

```
template <typename OutputIterator, typename A1, typename A2, typename A3>
struct grammar;
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputIterator</td>
<td>The underlying output iterator type that the rule is expected to work on.</td>
<td>none</td>
</tr>
<tr>
<td>A1, A2, A3</td>
<td>Either Signature, Delimiter or Locals in any order. See table below.</td>
<td>See table below.</td>
</tr>
</tbody>
</table>
Here is more information about the template parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>Specifies the grammar's synthesized (return value) and inherited attributes (arguments). More on this here: Nonterminal.</td>
<td>unused_type. When Signature defaults to unused_type, the effect is the same as specifying a signature of <code>void()</code> which is also equivalent to <code>unused_type()</code></td>
</tr>
<tr>
<td>Delimiter</td>
<td>Specifies the grammar's delimiter generator. Specify this if you want the grammar to delimit the generated output.</td>
<td>unused_type</td>
</tr>
<tr>
<td>Locals</td>
<td>Specifies the grammar's local variables. See Nonterminal.</td>
<td>unused_type</td>
</tr>
</tbody>
</table>

**Model of**

Nonterminal

**Notation**

g  A grammar

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in Nonterminal.

```cpp
template <typename OutputIterator>
struct my_grammar : grammar<OutputIterator, A1, A2, A3>
{
    my_grammar() : my_grammar
    { base_type(start, name)
        // Rule definitions
        start = /* ... */;
    }

    rule<OutputIterator, A1, A2, A3> start;
    // more rule declarations...
};
```

**Note**

The template parameters of a grammar and its start rule (the rule passed to the grammar's base class constructor) must match, otherwise you will see compilation errors.

**Attributes**

The generator attribute of the grammar is RT, its consumed attribute. See Attribute
Complexity

The complexity is defined by the complexity of the its definition.

Example

```
Note

The test harness for the example(s) below is presented in the Basics Examples section.
```

Some using declarations:

```cpp
using boost::spirit::ascii::space_type;
using boost::spirit::ascii::space;
using boost::spirit::int_;
using boost::spirit::karma::grammar;
using boost::spirit::karma::rule;
```

Basic grammar usage:

```cpp
struct num_list : grammar<output_iterator_type, space_type, std::vector<int> ()>
{
    num_list() : base_type(start)
    {
        using boost::spirit::int_; 
        num = int_;
        start = num << *(',' << num);
    }

    rule<output_iterator_type, space_type, std::vector<int> ()> start;
    rule<output_iterator_type, space_type, int() > num;
};
```

How to use the example grammar:

```cpp
num_list nlist;
std::vector<int> v;
v.push_back(123);
v.push_back(456);
v.push_back(789);

num_generator_attr_delim("123 , 456 , 789", nlist, space, v);
```

Numeric Generators

The library includes a couple of predefined objects for generating booleans, signed and unsigned integers, and real numbers. These generators are fully parametric. Most of the important aspects of numeric generation can be finely adjusted to suit. This includes the radix base, the exponent, the fraction etc. Policies control the real number generators’ behavior. There are some predefined policies covering the most common real number formats but the user can supply her own when needed.

The numeric parsers are fine tuned (employing loop unrolling and extensive template metaprogramming) with exceptional performance that rivals the low level C functions such as ltoa, sprintf, and _gcvt. Benchmarks reveal up to 2X speed over the C counterparts (see here: Performance of Numeric Generators). This goes to show that you can write extremely tight generic C++ code that rivals, if not surpasses C.
Module Header

```cpp
// forwards to <boost/spirit/home/karma/numeric.hpp>
#include <boost/spirit/include/karma_numeric.hpp>
```

Also, see Include Structure.

### Unsigned Integer Number Generators (\texttt{uint}, etc.)

#### Description

The \texttt{uint} class is the simplest among the members of the numerics package. The \texttt{uint} can generate unsigned integers of arbitrary length and size. The \texttt{uint} generator can be used to generate ordinary primitive C/C++ integers or even user defined scalars such as bigints (unlimited precision integers) if the type follows certain expression requirements (for more information about the requirements, see below). The \texttt{uint} is a template class. Template parameters fine tune its behavior.

#### Header

```cpp
// forwards to <boost/spirit/home/karma/numeric/uint.hpp>
#include <boost/spirit/include/karma_uint.hpp>
```

Also, see Include Structure.

#### Namespace

```cpp
Name

boost::spirit::lit // alias: boost::spirit::karma::lit
boost::spirit::bin // alias: boost::spirit::karma::bin
boost::spirit::oct // alias: boost::spirit::karma::oct
boost::spirit::hex // alias: boost::spirit::karma::hex
boost::spirit::ushort_ // alias: boost::spirit::karma::ushort_
boost::spirit::ulong_ // alias: boost::spirit::karma::ulong_
boost::spirit::uint_ // alias: boost::spirit::karma::uint_
boost::spirit::ulong_long // alias: boost::spirit::karma::ulong_long
```

#### Note

The generators \texttt{ulong_long} and \texttt{ulong_long(num)} are only available on platforms where the preprocessor constant \texttt{BOOST_HAS_LONG_LONG} is defined (i.e. on platforms having native support for unsigned \texttt{long long} (64 bit) unsigned integer types).

#### Note

\texttt{lit} is reused by the \texttt{String Generators}, the \texttt{Character Generators}, and the \texttt{Numeric Generators}. In general, a \texttt{char} generator is created when you pass in a character, a \texttt{string} generator is created when you pass in a string, and a \texttt{numeric} generator is created when you use a numeric literal.
Synopsis

```cpp
template <
    typename Num,
    unsigned Radix>
struct uint_generator;
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num</td>
<td>The numeric base type of the numeric generator.</td>
<td>unsigned int</td>
</tr>
<tr>
<td>Radix</td>
<td>The radix base. This can be any value in the (inclusive) range from 2 .. 36.</td>
<td>10</td>
</tr>
</tbody>
</table>

Model of

PrimitiveGenerator

Notation

- `num` Numeric literal, any unsigned integer value, or a Lazy Argument that evaluates to an unsigned integer value of type `Num`
- `Num` Type of `num`: any unsigned integer type, or in case of a Lazy Argument, its return value
- `Radix` An integer literal specifying the required radix for the output conversion. Valid values are from the (inclusive) range 2 .. 36.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveGenerator.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lit(num)</code></td>
<td>Generate the unsigned integer literal <code>num</code> using the default formatting (radix is 10). This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>ushort_</code>&lt;br&gt;<code>uint_</code>&lt;br&gt;<code>ulong_</code>&lt;br&gt;<code>ulong_long</code></td>
<td>Generate the unsigned integer provided by a mandatory attribute using the default formatting (radix is 10). This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>ushort_(num)</code>&lt;br&gt;<code>uint_(num)</code>&lt;br&gt;<code>ulong_(num)</code>&lt;br&gt;<code>ulong_long(num)</code></td>
<td>Generate the unsigned integer provided by the immediate literal value the generator is initialized from using the default formatting (radix is 10). If this generator has an associated attribute it succeeds only if the attribute is equal to the immediate literal (unless the underlying output stream reports an error). Otherwise this generator fails and does not generate any output.</td>
</tr>
<tr>
<td><code>bin</code>&lt;br&gt;<code>oct</code>&lt;br&gt;<code>hex</code></td>
<td>Generate the unsigned integer provided by a mandatory attribute using the default formatting and the corresponding radix (bin: radix is 2, oct: radix is 8, hex: radix is 16). This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>bin(num)</code>&lt;br&gt;<code>oct(num)</code>&lt;br&gt;<code>hex(num)</code></td>
<td>Generate the unsigned integer provided by the immediate literal value the generator is initialized from using the default formatting and the corresponding radix (bin: radix is 2, oct: radix is 8, hex: radix is 16). If this generator has an associated attribute it succeeds only if the attribute is equal to the immediate literal (unless the underlying output stream reports an error). Otherwise this generator fails and does not generate any output.</td>
</tr>
</tbody>
</table>

All generators listed in the table above (except `lit(num)`) are predefined specializations of the `uint_generator<Num, Radix>` basic unsigned integer number generator type described below. It is possible to directly use this type to create unsigned integer generators using a wide range of formatting options.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint_generator&lt;</code>&lt;br&gt;<code>Num, Radix</code>&lt;br&gt;<code>&gt;()</code></td>
<td>Generate the unsigned integer of type <code>Num</code> provided by a mandatory attribute using the specified <code>Radix</code> (allowed values are from the (inclusive) range from 2 .. 36, the default value is 10). This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>uint_generator&lt;</code>&lt;br&gt;<code>Num, Radix</code>&lt;br&gt;<code>&gt;()(num)</code></td>
<td>Generate the unsigned integer of type <code>Num</code> provided by the immediate literal value the generator is initialized from, using the specified <code>Radix</code> (allowed values are from the (inclusive) range from 2 .. 36, the default value is 10). If this generator has an associated attribute it succeeds only if the attribute is equal to the immediate literal (unless the underlying output stream reports an error). Otherwise this generator fails and does not generate any output.</td>
</tr>
</tbody>
</table>
Additional Requirements

The following lists enumerate the requirements which must be met in order to use a certain type Num to instantiate and use a
\texttt{uint\_generator<Num,\ Radix>}.

If \texttt{boost::is\_integral<Num>::value} is true the type Num must have defined:

- comparison operators for: \texttt{<, <=, ==, !>,\ and\ >=}
- numeric operators for: \texttt{+, -, /, *,\ and\ %}

If \texttt{boost::is\_integral<Num>::value} is false the type Num must have defined:

- comparison operators for: \texttt{<, <=, ==, !>,\ and\ >=}
- numeric operators for: \texttt{+, -, /, *,\ and\ %}

- helper functions implementing the interface and the semantics of: \texttt{std::fmod, std::pow, std::lround, std::ltruncate, std::floor,\ and\ std::ceil}. These need to be defined in a way so that they will be found using argument dependent lookup (ADL).
## Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>lit(num)</td>
<td>unused</td>
</tr>
<tr>
<td>ushort_</td>
<td>unsigned short, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>ushort_(num)</td>
<td>unsigned short, attribute is optional, if it is supplied, the generator compares the attribute with num and succeeds only if both are equal, failing otherwise.</td>
</tr>
<tr>
<td>uint_</td>
<td>unsigned int, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>uint_(num)</td>
<td>unsigned int, attribute is optional, if it is supplied, the generator compares the attribute with num and succeeds only if both are equal, failing otherwise.</td>
</tr>
<tr>
<td>ulong_</td>
<td>unsigned long, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>ulong_(num)</td>
<td>unsigned long, attribute is optional, if it is supplied, the generator compares the attribute with num and succeeds only if both are equal, failing otherwise.</td>
</tr>
<tr>
<td>ulong_long</td>
<td>unsigned long long, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>ulong_long(num)</td>
<td>unsigned long long, attribute is optional, if it is supplied, the generator compares the attribute with num and succeeds only if both are equal, failing otherwise.</td>
</tr>
<tr>
<td>bin</td>
<td>unsigned int, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>oct</td>
<td>unsigned int, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>hex</td>
<td>unsigned int, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>bin(num)</td>
<td>unsigned int, attribute is optional, if it is supplied, the generator compares the attribute with num and succeeds only if both are equal, failing otherwise.</td>
</tr>
<tr>
<td>oct(num)</td>
<td>unsigned int, attribute is optional, if it is supplied, the generator compares the attribute with num and succeeds only if both are equal, failing otherwise.</td>
</tr>
<tr>
<td>hex(num)</td>
<td>unsigned int, attribute is optional, if it is supplied, the generator compares the attribute with num and succeeds only if both are equal, failing otherwise.</td>
</tr>
<tr>
<td>uint_generator&lt; Num, Radix &gt;()</td>
<td>Num, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>uint_generator&lt; Num, Radix &gt;()(num)</td>
<td>Num, attribute is optional, if it is supplied, the generator compares the attribute with num and succeeds only if both are equal, failing otherwise.</td>
</tr>
</tbody>
</table>
**Note**

In addition to their usual attribute of type `Num` all listed generators accept an instance of a `boost::optional<Num>` as well. If the `boost::optional<>` is initialized (holds a value) the generators behave as if their attribute was an instance of `Num` and emit the value stored in the `boost::optional<>`.

Otherwise the generators will fail.

**Complexity**

$O(N)$, where $N$ is the number of digits needed to represent the generated integer number

**Example**

**Note**

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::uint_;
using boost::spirit::karma::lit;
```

Basic usage of an `uint` generator:

```cpp
test_generator("2", lit(2U));
test_generator("2", uint_(2));
test_generator_attr("2", uint_(2), 2);
test_generator_attr("", uint_(2), 3);  // fails (as 2 != 3)!
test_generator_attr("2", uint_, 2);
```

**Signed Integer Number Generators (int_, etc.)**

**Description**

The `int_generator` can generate signed integers of arbitrary length and size. This is almost the same as the `uint_generator`. The only difference is the additional task of generating the '+' or '-' sign preceding the number. The class interface is the same as that of the `uint_generator`.

The `int_generator` generator can be used to emit ordinary primitive C/C++ integers or even user defined scalars such as bigints (unlimited precision integers) if the type follows certain expression requirements (for more information about the requirements, see below).
Header

```cpp
// forwards to <boost/spirit/home/karma/numeric/int.hpp>
#include <boost/spirit/include/karma_int.hpp>
```

Also, see Include Structure.

Namespace

```cpp
namespace boost::spirit { 

// aliases

using namespace boost::spirit::karma;

// char

using boost::spirit::lit = boost::spirit::karma::lit;
using boost::spirit::short_ = boost::spirit::karma::short_
using boost::spirit::int_ = boost::spirit::karma::int_
using boost::spirit::long_ = boost::spirit::karma::long_
using boost::spirit::long_long = boost::spirit::karma::long_long;

```}

Note

The generators `long_long` and `long_long(num)` are only available on platforms where the preprocessor constant `BOOST_HAS_LONG_LONG` is defined (i.e. on platforms having native support for `long long` (64 bit) integer types).

Note

`lit` is reused by the String Generators, the Character Generators, and the Numeric Generators. In general, a char generator is created when you pass in a character, a string generator is created when you pass in a string, and a numeric generator is created when you use a numeric literal.

Synopsis

```cpp
template <
    typename T
, unsigned Radix
, bool force_sign>
struct int_generator;
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>The numeric base type of the numeric parser.</td>
<td>int</td>
</tr>
<tr>
<td>Radix</td>
<td>The radix base. This can be either 2: binary, 8: octal, 10: decimal and 16: hexadecimal.</td>
<td>10</td>
</tr>
<tr>
<td>force_sign</td>
<td>If true, all numbers will have a sign (space for zero)</td>
<td>false</td>
</tr>
</tbody>
</table>
Model of

**PrimitiveGenerator**

Notation

- **num**
  - Numeric literal, any signed integer value, or a *Lazy Argument* that evaluates to a signed integer value of type *Num*

- **Num**
  - Type of num: any signed integer type

- **Radix**
  - A constant integer literal specifying the required radix for the output conversion. Valid values are 2, 8, 10, and 16.

- **force_sign**
  - A constant boolean literal specifying whether the generated number should always have a sign (‘+’ for positive numbers, ‘-’ for negative numbers and ‘ ’ for zero).

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in *PrimitiveGenerator*.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lit(num)</strong></td>
<td>Generate the integer literal num using the default formatting (radix is 10, sign is only printed for negative literals). This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><strong>short_</strong> <strong>int_</strong> <strong>long_</strong> <strong>long_long</strong></td>
<td>Generate the integer provided by a mandatory attribute using the default formatting (radix is 10, sign is only printed for negative literals). This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><strong>short_(num)</strong> <strong>int_(num)</strong> <strong>long_(num)</strong> <strong>long_long(num)</strong></td>
<td>Generate the integer provided by the immediate literal value the generator is initialized from using the default formatting (radix is 10, sign is only printed for negative literals). If this generator has an associated attribute it succeeds only if the attribute is equal to the immediate literal (unless the underlying output stream reports an error). Otherwise this generator fails and does not generate any output.</td>
</tr>
</tbody>
</table>

All generators listed in the table above (except *lit(num)*) are predefined specializations of the **int_generator<Num, Radix, force_sign>** basic integer number generator type described below. It is possible to directly use this type to create integer generators using a wide range of formatting options.
### Expression

```
int_generator<
    Num, Radix, force_sign
>(())
```

### Semantics

Generate the integer of type `Num` provided by a mandatory attribute using the specified `Radix` (possible values are 2, 8, 10, and 16, the default value is 10). If `force_sign` is false (the default), a sign is only printed for negative literals. If `force_sign` is true, all numbers will be printed using a sign, i.e. `'- '` for negative numbers, `'+ '` for positive numbers, and `' ' `for zeros. This generator never fails (unless the underlying output stream reports an error).

```
int_generator<
    Num, Radix, force_sign
>(num)
```

Generate the integer of type `Num` provided by the immediate literal value the generator is initialized from, using the specified `Radix` (possible values are 2, 8, 10, and 16, the default value is 10). If `force_sign` is false (the default), a sign is only printed for negative literals. If `force_sign` is true, all numbers will be printed using a sign, i.e. `'- '` for negative numbers, `'+ '` for positive numbers, and `' ' `for zeros. If this generator has an associated attribute it succeeds only if the attribute is equal to the immediate literal (unless the underlying output stream reports an error). Otherwise this generator fails and does not generate any output.

### Additional Requirements

The following lists enumerate the requirements which must be met in order to use a certain type `Num` to instantiate and use a `int_generator<Num, Radix, force_sign>`.

If `boost::is_integral<Num>::value` is true the type `Num` must have defined:

- comparison operators for: `<`, `<=`, `==`, `!=`, `>`, and `>=`
- numeric operators for: `+`, `-`, `/`, `*`, `%`, and unary `-`

If `boost::is_integral<Num>::value` is false the type `Num` must have defined:

- comparison operators for: `<`, `<=`, `==`, `!=`, `>`, and `>=`
- numeric operators for: `+`, `-`, `/`, `*`, `%`, and unary `-`
- helper functions implementing the interface and the semantics of: `std::fmod`, `std::fabs`, `std::pow`, `std::lround`, `std::l trunc`, `std::floor`, and `std::ceil`. These need to be defined in a way so that they will be found using argument dependent lookup (ADL).
**Attributes**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lit(num)</code></td>
<td>unused</td>
</tr>
<tr>
<td><code>short_</code></td>
<td><code>short_</code> attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td><code>short_(num)</code></td>
<td><code>short_</code> attribute is optional, if it is supplied, the generator compares the attribute with <code>num</code> and succeeds only if both are equal, failing otherwise.</td>
</tr>
<tr>
<td><code>int_</code></td>
<td><code>int_</code> attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td><code>int_(num)</code></td>
<td><code>int_</code> attribute is optional, if it is supplied, the generator compares the attribute with <code>num</code> and succeeds only if both are equal, failing otherwise.</td>
</tr>
<tr>
<td><code>long_</code></td>
<td><code>long_</code> attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td><code>long_(num)</code></td>
<td><code>long_</code> attribute is optional, if it is supplied, the generator compares the attribute with <code>num</code> and succeeds only if both are equal, failing otherwise.</td>
</tr>
<tr>
<td><code>long_long</code></td>
<td><code>long long</code> attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td><code>long_long(num)</code></td>
<td><code>long long</code> attribute is optional, if it is supplied, the generator compares the attribute with <code>num</code> and succeeds only if both are equal, failing otherwise.</td>
</tr>
<tr>
<td><code>int_generator&lt; Num, Radix, force_sign ()</code></td>
<td>Num, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td><code>int_generator&lt; Num, Radix, force_sign ()(num)</code></td>
<td>Num, attribute is optional, if it is supplied, the generator compares the attribute with <code>num</code> and succeeds only if both are equal, failing otherwise.</td>
</tr>
</tbody>
</table>

**Note**

In addition to their usual attribute of type `Num` all listed generators accept an instance of a `boost::optional<Num>` as well. If the `boost::optional<>` is initialized (holds a value) the generators behave as if their attribute was an instance of `Num` and emit the value stored in the `boost::optional<>`. Otherwise the generators will fail.

**Complexity**

O(N), where N is the number of digits needed to represent the generated integer number
Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::int_
using boost::spirit::karma::lit;
```

Basic usage of an int_ generator:

```cpp
test_generator("-2", lit(-2));
test_generator("-2", int_(-2));
test_generator_attr("-2", int_(-2), -2);
test_generator_attr("", int_(-2), 3); // fails (as -2 != 3)!
test_generator_attr("-2", int_, -2);
```

Real Number Generators (float_, double_, etc.)

Description

The real_generator can generate real numbers of arbitrary length and size limited by its template parameter, Num. The numeric base type Num can be a user defined numeric type such as fixed_point (fixed point reals) and bignum (unlimited precision numbers) if the type follows certain expression requirements (for more information about the requirements, see below).

Header

```cpp
#include <boost/spirit/home/karma/numeric/real.hpp>
```

Also, see Include Structure.

Namespace

Name

```cpp
boost::spirit::lit // alias: boost::spirit::karma::lit
boost::spirit::float_ // alias: boost::spirit::karma::float_
boost::spirit::double_ // alias: boost::spirit::karma::double_
boost::spirit::long_double // alias: boost::spirit::karma::long_double
```
Note

lit is reused by the String Generators, the Character Generators, and the Numeric Generators. In general, a char generator is created when you pass in a character, a string generator is created when you pass in a string, and a numeric generator is created when you use a numeric literal.

Synopsis

```cpp
template <typename Num, typename RealPolicies>
struct real_generator;
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num</td>
<td>The type of the real number to generate.</td>
<td>double</td>
</tr>
<tr>
<td>RealPolicies</td>
<td>The policies to use while converting the real number.</td>
<td>real_policies&lt;Num&gt;</td>
</tr>
</tbody>
</table>

For more information about the type RealPolicies see below.

Model of

PrimitiveGenerator

Notation

num Numeric literal, any real number value, or a Lazy Argument that evaluates to a real number value of type Num

Num Type of num: any real number type

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveGenerator.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>lit(num)</td>
<td>Generate the real number literal ( \text{num} ) using the default formatting (no trailing zeros, fixed representation for numbers ( \text{fabs}(n) \leq 1e5 ) &amp;&amp; ( \text{fabs}(n) &gt; 1e-3 ), scientific representation otherwise, 3 fractional digits, sign is only printed for negative literals). This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>float_</td>
<td>Generate the real number provided by a mandatory attribute using the default formatting (no trailing zeros, fixed representation for numbers ( \text{fabs}(n) \leq 1e5 ) &amp;&amp; ( \text{fabs}(n) &gt; 1e-3 ), scientific representation otherwise, 3 fractional digits, sign is only printed for negative literals). This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td>double_</td>
<td>Generate the real number provided by the immediate literal value the generator is initialized from using the default formatting (no trailing zeros, fixed representation for numbers ( \text{fabs}(n) \leq 1e5 ) &amp;&amp; ( \text{fabs}(n) &gt; 1e-3 ), scientific representation otherwise, 3 fractional digits, sign is only printed for negative literals). If this generator has an associated attribute it succeeds only if the attribute is equal to the immediate literal (unless the underlying output stream reports an error). Otherwise this generator fails and does not generate any output.</td>
</tr>
<tr>
<td>long_double</td>
<td>Generate the real point number provided by the immediate literal value the generator is initialized from using the default formatting (no trailing zeros, fixed representation for numbers ( \text{fabs}(n) \leq 1e5 ) &amp;&amp; ( \text{fabs}(n) &gt; 1e-3 ), scientific representation otherwise, 3 fractional digits, sign is only printed for negative literals). If this generator has an associated attribute it succeeds only if the attribute is equal to the immediate literal (unless the underlying output stream reports an error). Otherwise this generator fails and does not generate any output.</td>
</tr>
</tbody>
</table>

All generators listed in the table above (except `lit(num)`) are predefined specializations of the `real_generator<\text{Num}, \text{RealPolicies}>` basic real number generator type described below. It is possible to directly use this type to create real number generators using a wide range of formatting options.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>real_generator&lt;</code></td>
<td>Generate the real number of type ( \text{Num} ) provided by a mandatory attribute using the specified ( \text{RealPolicies} ). This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>Num, \text{RealPolicies} &gt;()</code></td>
<td></td>
</tr>
<tr>
<td><code>real_generator&lt;</code></td>
<td>Generate the real number of type ( \text{Num} ) provided by the immediate literal value the generator is initialized from using the specified ( \text{RealPolicies} ). If this generator has an associated attribute it succeeds only if the attribute is equal to the immediate literal (unless the underlying output stream reports an error). Otherwise this generator fails and does not generate any output.</td>
</tr>
<tr>
<td><code>Num, \text{RealPolicies} &gt;()</code></td>
<td></td>
</tr>
</tbody>
</table>

**Additional Requirements**

The following list enumerates the requirements which must be met in order to use a certain type \( \text{Num} \) to instantiate a `real_generator<\text{Num}, \text{Policies}>`.

In order to be usable as the first template parameter for `real_generator<>` the type \( \text{Num} \) must have defined:

- comparison operators for: \(<, \leq, =, !=, >, \geq\)
- numeric operators for: \(+, -, /, *, \%\)
functions implementing the interface and the semantics of: `std::fmod`, `std::pow`, `std::log10`, `std::lround`, `std::ltrunc`, `std::modf`, `std::floor`, and `std::ceil`. These need to be defined in a way so that they will be found using argument dependent lookup (ADL).

- a valid specialization of the type `std::numeric_limits<Num>` allowing for numeric property inspection.

**Attributes**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lit(num)</code></td>
<td>unused</td>
</tr>
<tr>
<td><code>float_</code></td>
<td><code>float_</code>, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td><code>float_(num)</code></td>
<td><code>float_</code>, attribute is optional, if it is supplied, the generator compares the attribute with <code>num</code> and succeeds only if both are equal, failing otherwise.</td>
</tr>
<tr>
<td><code>double_</code></td>
<td><code>double_</code>, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td><code>double_(num)</code></td>
<td><code>double_</code>, attribute is optional, if it is supplied, the generator compares the attribute with <code>num</code> and succeeds only if both are equal, failing otherwise.</td>
</tr>
<tr>
<td><code>long_double</code></td>
<td><code>long double</code>, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td><code>long_double(num)</code></td>
<td><code>long double</code>, attribute is optional, if it is supplied, the generator compares the attribute with <code>num</code> and succeeds only if both are equal, failing otherwise.</td>
</tr>
<tr>
<td><code>real_generator&lt; Num, Policies &gt;()</code></td>
<td><code>Num</code>, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td><code>real_generator&lt; Num, Policies &gt;()(num)</code></td>
<td><code>Num</code>, attribute is optional, if it is supplied, the generator compares the attribute with <code>num</code> and succeeds only if both are equal, failing otherwise.</td>
</tr>
</tbody>
</table>

**Note**

In addition to their usual attribute of type `Num` all listed generators accept an instance of a `boost::optional<Num>` as well. If the `boost::optional<>` is initialized (holds a value) the generators behave as if their attribute was an instance of `Num` and emit the value stored in the `boost::optional<>`. Otherwise the generators will fail.

**Real Number Formatting Policies**

If special formatting of a real number is needed, overload the policy class `real_policies<Num>` and use it as a template parameter to the `real_generator<>` real number generator. For instance:
// define a new real number formatting policy
template <typename Num>
struct scientific_policy : real_policies<Num>
{
    // we want the numbers always to be in scientific format
    static int floatfield(Num n) { return fmtflags::scientific; }
};

// define a new generator type based on the new policy
typedef real_generator<double, scientific_policy<double> > science_type;

science_type const scientific = science_type();

// use the new generator
generate(sink, science_type(), 1.0); // will output: 1.0e00
generate(sink, scientific, 0.1); // will output: 1.0e-01

The template parameter Num should be the type to be formatted using the overloaded policy type. At the same time Num will be used as the attribute type of the created real number generator.

**Real Number Formatting Policy Expression Semantics**

A real number formatting policy should expose the following variables and functions:
### Description

This is the main function used to generate the output for a real number. It is called by the real generator in order to perform the conversion. In theory all of the work can be implemented here, but the easiest way is to use existing functionality provided by the type specified by the template parameter `Inserter`. The default implementation of this function is:

```cpp
template <typename Inserter, typename OutputIterator, typename Policies>
bool call (OutputIterator& sink, Num n, Policies const& p);
```

`sink` is the output iterator to use for generation

*n* is the real number to convert

*p* is the instance of the policy type used to instantiate this real number generator.

The default behavior is to not to require generating a sign. If the function `force_sign()` returns true, then all generated numbers will have a sign ('+' or '-', zeros will have a space instead of a sign).

*n* is the real number to output. This can be used to adjust the required behavior depending on the value of this number.

Return whether trailing zero digits have to be emitted in the fractional part of the output. If set, this flag instructs the real number generator to emit trailing zeros up to the required precision digits (as returned by the `precision()` function).

*n* is the real number to output. This can be used to adjust the required behavior depending on the value of this number.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
</table>
| `template <typename Inserter, typename OutputIterator, typename Policies>
bool call (OutputIterator& sink, Num n, Policies const& p);` | This is the main function used to generate the output for a real number. It is called by the real generator in order to perform the conversion. In theory all of the work can be implemented here, but the easiest way is to use existing functionality provided by the type specified by the template parameter `Inserter`. The default implementation of this functions is: |
| `template <typename Inserter, typename OutputIterator, typename Policies>
static bool call (OutputIterator& sink, Num n, Policies const& p)
{` | `return Inserter::call_n(sink, n, p);` |
<p>| <code>bool force_sign(Num n);</code> | The default behavior is to not to require generating a sign. If the function <code>force_sign()</code> returns true, then all generated numbers will have a sign ('+' or '-', zeros will have a space instead of a sign). |
| <code>bool trailing_zeros(Num n);</code> | Return whether trailing zero digits have to be emitted in the fractional part of the output. If set, this flag instructs the real number generator to emit trailing zeros up to the required precision digits (as returned by the <code>precision()</code> function). |</p>
<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int floatfield(Num n);</code></td>
<td>Decide, which representation type to use in the generated output. By default all numbers having an absolute value of zero or in between 0.001 and 100000 will be generated using the fixed format, all others will be generated using the scientific representation. The <code>trailing_zeros()</code> can be used to force the output of trailing zeros in the fractional part up to the number of digits returned by the <code>precision()</code> member function. The default is not to generate the trailing zeros. n is the real number to output. This can be used to adjust the formatting flags depending on the value of this number. The return value has to be either <code>fmtflags::scientific</code> (generate real number values in scientific notation) or <code>fmtflags::fixed</code> (generate real number values in fixed-point notation).</td>
</tr>
<tr>
<td><code>unsigned precision(Num n);</code></td>
<td>Return the maximum number of decimal digits to generate in the fractional part of the output. n is the real number to output. This can be used to adjust the required precision depending on the value of this number. If the trailing zeros flag is specified the fractional part of the output will be 'filled' with zeros, if appropriate. <strong>Note:</strong> If the trailing_zeros flag is not in effect additional semantics apply. See the description for the <code>fraction_part()</code> function below. Moreover, this precision will be limited to the value of <code>std::numeric_limits&lt;T&gt;::digits10 + 1</code>.</td>
</tr>
<tr>
<td><code>template &lt;bool ForceSign, typename OutputIterator&gt; bool integer_part(OutputIterator&amp; sink, Num n, bool sign, bool force_sign);</code></td>
<td>This function is called to generate the integer part of the real number. sink is the output iterator to use for generation n is the absolute value of the integer part of the real number to convert (always non-negative) sign is the sign of the overall real number to convert. force_sign is a flag indicating whether a sign has to be generated even for non-negative numbers (this is the same as has been returned from the function <code>force_sign()</code> described above) The return value defines the outcome of the whole generator. If it is <code>false</code>, no further output is generated, immediately returning <code>false</code> from the calling <code>real_generator</code> as well. If it is <code>true</code>, normal output generation continues.</td>
</tr>
<tr>
<td>Expression</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>\begin{verbatim}template &lt;typename OutputIterator&gt; bool dot(OutputIterator&amp; sink, Num n, unsigned precision);\end{verbatim}</td>
<td>This function is called to generate the decimal point. sink is the output iterator to use for generation.</td>
</tr>
</tbody>
</table>

n is the fractional part of the real number to convert. Note that this number is scaled such, that it represents the number of units which correspond to the value returned from the \texttt{precision()} function earlier. I.e. a fractional part of 0.01234 is represented as 1234 when the function \texttt{precision()} returned 5. precision is the number of digits to emit as returned by the function \texttt{precision()} described above. This is given to allow to decide, whether a decimal point has to be generated at all. **Note:** If the \texttt{trailing_zeros} flag is not in effect additional comments apply. See the description for the \texttt{fraction_part()} function below. The return value defines the outcome of the whole generator. If it is false, no further output is generated, immediately returning false from the calling \texttt{real_generator} as well. If it is true, normal output generation continues. |
<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;typename OutputIterator&gt; bool fraction_part(OutputIterator&amp; sink, Num n, unsigned adjprec, unsigned precision);</td>
<td>This function is called to generate the fractional part of the number.</td>
</tr>
<tr>
<td></td>
<td>sink is the output iterator to use for generation</td>
</tr>
<tr>
<td></td>
<td>n is the fractional part of the real number to convert. Note that this number is scaled such, that it represents the number of units which correspond to the value returned from the precision() function earlier. I.e. a fractional part of 0.01234 is represented as 1234 when the function precision() returned 5.</td>
</tr>
<tr>
<td></td>
<td>adjprec is the corrected number of digits to emit (see note below)</td>
</tr>
<tr>
<td></td>
<td>precision is the number of digits to emit as returned by the function precision() described above</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> If trailing_zeros() returns false the adjprec parameter will have been corrected from the value the precision() function returned earlier (defining the maximal number of fractional digits) in the sense, that it takes into account trailing zeros. I.e. a real number 0.0123 and a value of 5 returned from precision() will result in:</td>
</tr>
<tr>
<td></td>
<td>trailing_zeros() returned false: n will be 123, and adjprec will be 4 (as we need to print 0123)</td>
</tr>
<tr>
<td></td>
<td>trailing_zeros() returned true: n will be 1230, and adjprec will be 5 (as we need to print 01230)</td>
</tr>
<tr>
<td></td>
<td>The missing preceding zeros in the fractional part have to be supplied by the implementation of this policy function.</td>
</tr>
<tr>
<td></td>
<td>The return value defines the outcome of the whole generator. If it is false, no further output is generated, immediately returning false from the calling real_generator as well. If it is true, normal output generation continues.</td>
</tr>
<tr>
<td>template &lt;typename CharEncoding, typename Tag, typename OutputIterator&gt; bool exponent(OutputIterator&amp; sink, long n);</td>
<td>This function is called to generate the exponential part of the number (this is called only if the floatfield() function returned the fmtflags::scientific flag).</td>
</tr>
<tr>
<td></td>
<td>sink is the output iterator to use for generation</td>
</tr>
<tr>
<td></td>
<td>n is the (signed) exponential part of the real number to convert.</td>
</tr>
<tr>
<td></td>
<td>The template parameters CharEncoding and Tag are either of the type unused_type or describe the character class and conversion to be applied to any output possibly influenced by either the lower() or upper() directives.</td>
</tr>
<tr>
<td></td>
<td>The return value defines the outcome of the whole generator. If it is false, no further output is generated, immediately returning false from the calling real_generator as well. If it is true, normal output generation continues.</td>
</tr>
</tbody>
</table>
### Expression

```cpp
template <typename CharEncoding
        , typename Tag, typename OutputIterator>
bool nan (OutputIterator& sink, Num n
            , bool force_sign);
```

### Description

This function is called whenever the number to print is a non-normal real number of type NaN.

- **sink** is the output iterator to use for generation
- **n** is the (signed) real number to convert
- **force_sign** is a flag indicating whether a sign has to be generated even for non-negative numbers (this is the same as has been returned from the function `force_sign()` described above)

The template parameters `CharEncoding` and `Tag` are either of the type `unused_type` or describe the character class and conversion to be applied to any output possibly influenced by either the `lower[]` or `upper[]` directives.

The return value defines the outcome of the whole generator. If it is `false`, no further output is generated, immediately returning `false` from the calling `real_generator` as well. If it is `true`, normal output generation continues.

### Expression

```cpp
template <typename CharEncoding
        , typename Tag, typename OutputIterator>
bool inf (OutputIterator& sink, Num n
            , bool force_sign);
```

### Description

This function is called whenever the number to print is a non-normal real number of type Inf.

- **sink** is the output iterator to use for generation
- **n** is the (signed) real number to convert
- **force_sign** is a flag indicating whether a sign has to be generated even for non-negative numbers (this is the same as has been returned from the function `force_sign()` described above)

The template parameters `CharEncoding` and `Tag` are either of the type `unused_type` or describe the character class and conversion to be applied to any output possibly influenced by either the `lower[]` or `upper[]` directives.

The return value defines the outcome of the whole generator. If it is `false`, no further output is generated, immediately returning `false` from the calling `real_generator` as well. If it is `true`, normal output generation continues.

---

**Tip**

The easiest way to implement a proper real number formatting policy is to derive a new type from the the type `real_policies<>` while overriding the aspects of the formatting which need to be changed.

**Complexity**

O(N), where N is the number of digits needed to represent the generated real number.
Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::double_;  
using boost::spirit::karma::lit;
```

Basic usage of a double generator:

```cpp
test_generator("2.0", lit(2.0));  
test_generator("2.0", double_(2));  
test_generator_attr("2.0", double_(2.0), 2.0);  
test_generator_attr("", double_(2.0), 3.0);  // fails (as 2.0 != 3.0)!

test_generator_attr("-2.0", double_,-2.0);  
test_generator_attr("1.234e05", double_ ,1234.0e2);  
test_generator_attr("1.234e-06", double_ ,0.000001234);
```

Boolean Generators (bool_)

Description

As you might expect, the bool_generator can generate output from boolean values. The bool_generator generator can be used to generate output from ordinary primitive C/C++ bool values or user defined boolean types if the type follows certain expression requirements (for more information about the requirements, see below). The bool_generator is a template class. Template parameters fine tune its behavior.

Header

```cpp
   // forwards to <boost/spirit/home/karma/numeric/bool.hpp>  
   #include <boost/spirit/include/karma/numeric/bool.hpp>
```

Also, see Include Structure.
Namespace

Name

boost::spirit::lit // alias: boost::spirit::karma::lit
boost::spirit::bool_ // alias: boost::spirit::karma::bool_
boost::spirit::true_ // alias: boost::spirit::karma::true_
boost::spirit::false_ // alias: boost::spirit::karma::false_

Note

lit is reused by the String Generators, the Character Generators, and the Numeric Generators. In general, a char
generator is created when you pass in a character, a string generator is created when you pass in a string, and a nu-
meric generator is created when you use a numeric (boolean) literal.

Synopsis

```cpp
template <
    typename B,
    unsigned Policies>
struct bool_generator;
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>The boolean base type of the boolean generator.</td>
<td>bool</td>
</tr>
<tr>
<td>Policies</td>
<td>The policies to use while converting the boolean.</td>
<td>bool_policies&lt;B&gt;</td>
</tr>
</tbody>
</table>

Model of

PrimitiveGenerator

Notation

- Boolean literal, or a Lazy Argument that evaluates to a boolean value of type B
- Type of b: any type usable as a boolean, or in case of a Lazy Argument, its return value

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveGenerator.
## Expression | Semantics
---|---
\texttt{lit(b)} | Generate the boolean literal \(b\) using the default formatting (false is generated as "false", and true is generated as "true"). This generator never fails (unless the underlying output stream reports an error).  
\texttt{bool_} | Generate the boolean value provided by a mandatory attribute using the default formatting (false is generated as "false", and true is generated as "true"). This generator never fails (unless the underlying output stream reports an error).  
\texttt{bool_}(b) | Generate the boolean value provided by the immediate literal value the generator is initialized from using the default formatting (false is generated as "false", and true is generated as "true"). If this generator has an associated attribute it succeeds only if the attribute is equal to the immediate literal (unless the underlying output stream reports an error). Otherwise this generator fails and does not generate any output.  
\texttt{true_} | Generate "true". If this generator has an associated attribute it succeeds only if the attribute is \texttt{true} as well (unless the underlying output stream reports an error).  
\texttt{false_} | Generate "false". If this generator has an associated attribute it succeeds only if the attribute is \texttt{false} as well (unless the underlying output stream reports an error).  

All generators listed in the table above (except \texttt{lit(num)}\texttt{)} are predefined specializations of the \texttt{bool_generator<B, Policies> basic boolean generator type described below. It is possible to directly use this type to create boolean generators using a wide range of formatting options.

## Expression | Semantics
---|---
\texttt{bool_generator<}
\texttt{B, Policies}
\texttt{>()} | Generate the boolean of type \(B\) provided by a mandatory attribute using the specified Policies. This generator never fails (unless the underlying output stream reports an error).  
\texttt{bool_generator<}
\texttt{B, Policies}
\texttt{>()(b)} | Generate the boolean of type \(B\) provided by the immediate literal value the generator is initialized from, using the specified Policies. If this generator has an associated attribute it succeeds only if the attribute is equal to the immediate literal (unless the underlying output stream reports an error). Otherwise this generator fails and does not generate any output.  

### Note

All boolean generators properly respect the \texttt{upper} and \texttt{lower} directives.

### Additional Requirements

The following lists enumerate the requirements which must be met in order to use a certain type \(B\) to instantiate and use a \texttt{bool_generator<B, Policies>}.  

338
The type B:

- must be (safely) convertible to bool

**Attributes**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool_(b)</td>
<td>unused</td>
</tr>
<tr>
<td>bool_</td>
<td>bool, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>bool_(b)</td>
<td>bool, attribute is optional, if it is supplied, the generator compares the attribute with b and succeeds only if both are equal, failing otherwise.</td>
</tr>
<tr>
<td>bool_generator&lt; B, Policies &gt;()</td>
<td>B, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>bool_generator&lt; B, Policies &gt;()(b)</td>
<td>B, attribute is optional, if it is supplied, the generator compares the attribute with b and succeeds only if both are equal, failing otherwise.</td>
</tr>
</tbody>
</table>

**Note**

In addition to their usual attribute of type B all listed generators accept an instance of a boost::optional<B> as well. If the boost::optional<> is initialized (holds a value) the generators behave as if their attribute was an instance of B and emit the value stored in the boost::optional<>. Otherwise the generators will fail.

**Boolean Formatting Policies**

If special formatting of a boolean is needed, overload the policy class bool_policies<B> and use it as a template parameter to the bool_generator<> boolean generator. For instance:

```cpp
struct special_bool_policy : karma::bool_policies<> {
    template <typename CharEncoding, typename Tag
              , typename OutputIterator>
    static bool generate_false(OutputIterator& sink, bool b)
    {
        // we want to spell the names of false as eurt (true backwards)
        return string_inserter<CharEncoding, Tag>::call(sink, "eurt");
    }
};

typedef karma::bool_generator<special_bool_policy> backwards_bool_type;
backwards_bool_type const backwards_bool;
```

The template parameter B should be the type to be formatted using the overloaded policy type. At the same time B will be used as the attribute type of the created real number generator. The default for B is bool.
**Boolean Formatting Policy Expression Semantics**

A boolean formatting policy should expose the following:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;typename Inserter, typename OutputIterator, typename Policies&gt;</td>
<td>This is the main function used to generate the output for a boolean. It is called by the boolean generator in order to perform the conversion. In theory all of the work can be implemented here, but the easiest way is to use existing functionality provided by the type specified by the template parameter Inserter. The default implementation of this functions is:</td>
</tr>
<tr>
<td>bool call (OutputIterator&amp; sink, Num n, Policies const&amp; p);</td>
<td></td>
</tr>
<tr>
<td>template &lt;typename CharEncoding, typename Tag, typename OutputIterator&gt;</td>
<td>This function is called to generate the boolean if it is false.</td>
</tr>
<tr>
<td>bool generate_false(OutputIterator&amp; sink, B b);</td>
<td></td>
</tr>
<tr>
<td>template &lt;typename CharEncoding, typename Tag, typename OutputIterator&gt;</td>
<td>This function is called to generate the boolean if it is true.</td>
</tr>
<tr>
<td>bool generate_true(OutputIterator&amp; sink, B b);</td>
<td></td>
</tr>
</tbody>
</table>

**Complexity**

O(N), where N is the number of characters needed to represent the generated boolean.
**Example**

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::bool_;  
using boost::spirit::karma::lit;
```

Basic usage of an `bool_` generator:

```cpp
test_generator("true", lit(true));
test_generator("false", bool_(false));
test_generator_attr("true", bool_(true), true);
test_generator_attr("false", bool_, false);  // fails (as true != false)!
test_generator_attr("false", bool_, false);
```

**Generator Operators**

Operators are used as a means for object composition and embedding. Simple generators may be composed to form composites through operator overloading, crafted to approximate the syntax of Parsing Expression Grammar (PEG). An expression such as:

```
a | b
```

yields a new generator type which is a composite of its operands, a and b.

This module includes different generators which get instantiated if one of the overloaded operators is used with more primitive generator constructs. It includes sequences (a << b), alternatives (a | b), Kleene star (unary *), plus (unary +), optional (unary -), lists (a % b), and the two predicates, the *and* predicate (unary &) and the *not* predicate (unary !).

**Module Header**

```cpp
// forwards to <boost/spirit/home/karma/operator.hpp>
#include <boost/spirit/home/karma/operator.hpp>
```

Also, see Include Structure.

**Sequence Generator (a << b)**

**Description**

Generator sequences are used to consecutively combine different, more primitive generators. All generators in a sequence are invoked from left to right as long as they succeed.
Model of NaryGenerator

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in NaryGenerator.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt;&lt; b</td>
<td>The generators a and b are executed sequentially from left to right and as long as they succeed. A failed generator stops the execution of the entire sequence and makes the sequence fail as well.</td>
</tr>
</tbody>
</table>

It is important to note, that sequences don’t perform any buffering of the output generated by its elements. That means that any failing sequence might have already generated some output, which is not rolled back.

Tip

The simplest way to force a sequence to behave as if it did buffering is to wrap it into a buffering directive (see buffer):

```
buffer[a << b << c]
```

which will not generate any output in case of a failing sequence.

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt;&lt; b (sequence)</td>
<td>a: A, b: B --&gt; (a &lt;&lt; b): tuple&lt;A, B&gt;</td>
</tr>
<tr>
<td></td>
<td>a: A, b: Unused --&gt; (a &lt;&lt; b): A</td>
</tr>
<tr>
<td></td>
<td>a: Unused, b: B --&gt; (a &lt;&lt; b): B</td>
</tr>
<tr>
<td></td>
<td>a: Unused, b: Unused --&gt; (a &lt;&lt; b): Unused</td>
</tr>
<tr>
<td></td>
<td>a: A, b: A --&gt; (a &lt;&lt; b): vector&lt;A&gt;</td>
</tr>
<tr>
<td></td>
<td>a: vector&lt;A&gt;, b: A --&gt; (a &lt;&lt; b): vector&lt;A&gt;</td>
</tr>
<tr>
<td></td>
<td>a: A, b: vector&lt;A&gt; --&gt; (a &lt;&lt; b): vector&lt;A&gt;</td>
</tr>
<tr>
<td></td>
<td>a: vector&lt;A&gt;, b: vector&lt;A&gt; --&gt; (a &lt;&lt; b): vector&lt;A&gt;</td>
</tr>
</tbody>
</table>
Important

The table above uses \texttt{tuple\langle A, B \rangle} and \texttt{vector\langle A \rangle} as placeholders only.

The notation \texttt{tuple\langle A, B \rangle} stands for \textit{any fusion sequence of two elements}, where \( A \) is the type of its first element and \( B \) is the type of its second element.

The notation of \texttt{vector\langle A \rangle} stands for \textit{any STL container} holding elements of type \( A \).

The attribute composition and propagation rules as shown in the table above make sequences somewhat special as they can operate in two modes if all elements have the same attribute type: consuming fusion sequences and consuming STL containers. The selected mode depends on the type of the attribute supplied.

Complexity

The overall complexity of the sequence generator is defined by the sum of the complexities of its elements. The complexity of the sequence itself is \( O(N) \), where \( N \) is the number of elements in the sequence.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::double_;  
```

Basic usage of a sequence:

```cpp
test_generator_attr("1.0,2.0", double_ << ',' << double_, std::make_pair(1.0, 2.0));
```

Alternative Generator \((a \mid b)\)

Description

Generator alternatives are used to combine different, more primitive generators into alternatives. All generators in an alternative are invoked from left to right until one of them succeeds.

Header

```cpp
// forwards to <boost/spirit/home/karma/operator/alternative.hpp>
#include <boost/spirit/include/karma_alternative.hpp>
```

Also, see Include Structure.
Model of

NaryGenerator

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in NaryGenerator.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

Alternatives intercept and buffer the output of the currently executed element. This allows to avoid partial outputs from failing elements as the buffered content will be forwarded to the actual output only after an element succeeded.

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b (alternative)</td>
</tr>
<tr>
<td>a</td>
<td>b (alternative)</td>
</tr>
<tr>
<td>a</td>
<td>b (alternative)</td>
</tr>
<tr>
<td>a</td>
<td>b (alternative)</td>
</tr>
<tr>
<td>a</td>
<td>b (alternative)</td>
</tr>
</tbody>
</table>

Important

The table above uses variant<A, B> as a placeholder only. The notation variant<A, B> stands for the type boost::variant<A, B>.

The attribute handling of Alternatives is special as their behavior is not completely defined at compile time. First of all the selected alternative element depends on the actual type of the attribute supplied to the alternative generator (i.e. what is stored in the variant). The attribute type supplied at runtime narrows the set of considered alternatives to those being compatible attribute wise. The remaining alternatives are tried sequentially until the first of them succeeds. See below for an example of this behavior.

Complexity

The overall complexity of the alternative generator is defined by the sum of the complexities of its elements. The complexity of the alternative itself is O(N), where N is the number of elements in the alternative.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:
Some using declarations:

```cpp
using boost::spirit::karma::double_;
using boost::spirit::karma::ascii::string;
```

Basic usage of an alternative. While being only the second alternative, the `double_` generator is chosen for output formatting because the supplied attribute type is not compatible (i.e. not convertible) to the attribute type of the `string` alternative.

```cpp
boost::variant<std::string, double> v1(1.0);
test_generator_attr("1.0", string | double_, v1);
test_generator_attr("2.0", string | double_, 2.0);
```

The same formatting rules may be used to output a string. This time we supply the string "example", resulting in the first alternative to be chosen for the generated output.

```cpp
boost::variant<std::string, double> v2("example");
test_generator_attr("example", string | double_, v2);
test_generator_attr("example", string | double_, "example");
```

**Kleene Star Generator (**\*a\*)**

**Description**

Kleene star generators are used to repeat the execution of an embedded generator zero or more times. Regardless of the success of the embedded generator, the Kleene star generator always succeeds.

**Header**

```cpp
#include <boost/spirit/home/karma/operator/kleene.hpp>
```

Also, see Include Structure.

**Model of**

`UnaryGenerator`

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in `UnaryGenerator`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>*a</td>
<td>The generator a is executed zero or more times depending on the availability of an attribute. The execution of a stops after the attribute values passed to the Kleene star generator are exhausted. The Kleene star always succeeds (unless the underlying output stream reports an error).</td>
</tr>
</tbody>
</table>
**Note**

All failing iterations of the embedded generator will consume one element from the supplied attribute.

**Attributes**

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
</table>
| *\(a\) \(\text{Kleene star, unary }^*\) | \(a: A \rightarrow *a: \text{vector}\langle A\rangle\)  \\
|           | \(a: \text{Unused} \rightarrow *a: \text{Unused}\) |

**Important**

The table above uses \(\text{vector}\langle A\rangle\) as a placeholder only. The notation of \(\text{vector}\langle A\rangle\) stands for any STL container holding elements of type \(A\).

The Kleene star generator will execute its embedded generator once for each element in the provided container attribute as long as the embedded generator succeeds. On each iteration it will pass the next consecutive element from the container attribute to the embedded generator. Therefore the number of iterations will not be larger than the number of elements in the container passed as its attribute. An empty container will make the Kleene star generate no output at all.

It is important to note, that the Kleene star does not perform any buffering of the output generated by its embedded elements. That means that any failing element generator might have already generated some output, which is not rolled back.

**Tip**

The simplest way to force a Kleene star to behave as if it did buffering is to wrap it into a buffering directive (see buffer):

```
buffer[\(*a\)]
```

which will not generate any output in case of a failing generator \(*a\). The expression:

```
\(*(buffer[a])\)
```

will not generate any partial output from a generator \(a\) if it fails generating in the middle of its output. The overall expression will still generate the output as produced by all successful invocations of the generator \(a\).

**Complexity**

The overall complexity of the Kleene star generator is defined by the complexity of its embedded generator multiplied by the number of executed iterations. The complexity of the Kleene star itself is \(O(N)\), where \(N\) is the number of elements in the container passed as its attribute.
Example

Note
The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::double_;  
using boost::spirit::karma::space;
```

Basic usage of a Kleene star generator:

```cpp
std::vector<double> v;
v.push_back(1.0);          
v.push_back(2.0);          
v.push_back(3.0);          
test_generator_attr_delim("1.0 2.0 3.0 ", *double_, space, v);
```

Plus Generator (+a)

Description
The Plus generator is used to repeat the execution of an embedded generator one or more times. It succeeds if the embedded generator has been successfully executed at least once.

Header

```cpp
// forwards to <boost/spirit/home/karma/operator/plus.hpp>
#include <boost/spirit/include/karma_plus.hpp>
```

Also, see Include Structure.

Model of

UnaryGenerator

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryGenerator.
The generator \( a \) is executed one or more times depending on the availability of an attribute. The execution of \( a \) stops after the attribute values passed to the plus generator are exhausted. The plus generator succeeds as long as its embedded generator has been successfully executed at least once (unless the underlying output stream reports an error).

Note

All failing iterations of the embedded generator will consume one element from the supplied attribute. The overall \( +a \) will succeed as long as at least one invocation of the embedded generator will succeed (unless the underlying output stream reports an error).

Attributes

See Compound Attribute Notation.

Expression | Attribute
---|---
\( +a \) (unary \(+\)) | \( a: A \to +a: \text{vector}\langle A\rangle \)
| \( a: \text{Unused} \to +a: \text{Unused} \)

Important

The table above uses \( \text{vector}\langle A\rangle \) as a placeholder only. The notation of \( \text{vector}\langle A\rangle \) stands for any STL container holding elements of type \( A \).

The Plus generator will execute its embedded generator once for each element in the provided container attribute as long as the embedded generator succeeds. On each iteration it will pass the next consecutive element from the container attribute to the embedded generator. Therefore the number of iterations will not be larger than the number of elements in the container passed as its attribute. An empty container will make the plus generator fail.

It is important to note, that the plus generator does not perform any buffering of the output generated by its embedded elements. That means that any failing element generator might have already generated some output, which is not rolled back.

Tip

The simplest way to force a plus generator to behave as if it did buffering is to wrap it into a buffering directive (see \texttt{buffer}):

\[
\text{buffer}[+a]
\]

which will not generate any output in case of a failing generator \( +a \). The expression:

\[
+(\text{buffer}[a])
\]

will not generate any partial output from a generator \( a \) if it fails generating in the middle of its output. The overall expression will still generate the output as produced by all successful invocations of the generator \( a \).
Complexity

The overall complexity of the plus generator is defined by the complexity of its embedded generator multiplied by the number of executed iterations. The complexity of the plus generator itself is $O(N)$, where $N$ is the number of elements in the container passed as its attribute.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::double_;  
using boost::spirit::karma::space;
```

Basic usage of a plus generator:

```cpp
std::vector<double> v1;
v1.push_back(1.0);
v1.push_back(2.0);
v1.push_back(3.0);
test_generator_attr_delim("1.0 2.0 3.0 ", +double_, space, v1);
```

A more sophisticated use case showing how to leverage the fact that plus is failing for empty containers passed as its attribute:

```cpp
std::vector<double> v2;  // empty container
test_generator_attr("empty", +double_ | "empty", v2);
```

List Generator ($a \% b$)

Description

The list generator is used to repeat the execution of an embedded generator and intersperse it with the output of another generator one or more times. It succeeds if the embedded generator has been successfully executed at least once.

Header

```cpp
// forwards to <boost/spirit/home/karma/operator/list.hpp>
#include <boost/spirit/include/karma_list.hpp>
```

Also, see Include Structure.
Model of

**BinaryGenerator**

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in **BinaryGenerator**.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>a % b</td>
<td>The generator a is executed one or more times depending on the availability of an attribute. The output generated by a is interspersed with the output generated by b. The list generator succeeds if its first embedded generator has been successfully executed at least once (unless the underlying output stream reports an error).</td>
</tr>
</tbody>
</table>

The list expression a % b is a shortcut for a << *(b << a). It is almost semantically equivalent, except for the attribute of b, which gets ignored in the case of the list generator.

**Note**

All failing iterations of the embedded generator will consume one element from the supplied attribute. The overall a % b will succeed as long as at least one invocation of the embedded generator, a, will succeed (unless the underlying output stream reports an error).

**Attributes**

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
</table>
| a % b (list) | a: A, b: B --> (a % b): vector<A>  
               a: Unused, b: B --> (a % b): Unused |

**Important**

The table above uses vector<A> as a placeholder only. The notation of vector<A> stands for any STL container holding elements of type A.

The list generator will execute its embedded generator once for each element in the provided container attribute and as long as the embedded generator succeeds. The output generated by its first generator will be interspersed by the output generated by the second generator. On each iteration it will pass the next consecutive element from the container attribute to the first embedded generator. The second embedded generator does not get passed any attributes (it gets invoked using an unused_type as its attribute). Therefore the number of iterations will not be larger than the number of elements in the container passed as its attribute. An empty container will make the list generator fail.
Tip

If you want to use the list generator and still allow for an empty attribute, you can use the optional operator (see Optional (unary `-`)):

```
-(a % b)
```

which will succeed even if the provided container attribute does not contain any elements.

Complexity

The overall complexity of the list generator is defined by the complexity of its embedded generators multiplied by the number of executed iterations. The complexity of the list generator itself is $O(N)$, where $N$ is the number of elements in the container passed as its attribute.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::double_;  
```

Basic usage of a list generator:

```cpp
std::vector<double> v1;
v1.push_back(1.0);
test_generator_attr("1.0", double_ % ',', v1);
```

Optional Generator (\(-a\))

Description

The optional generator is used to conditionally execute an embedded generator. It succeeds always.
Header

```cpp
// forwards to <boost/spirit/home/karma/operator/optional.hpp>
#include <boost/spirit/include/karma_optional.hpp>
```

Also, see Include Structure.

Model of

`UnaryGenerator`

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in `UnaryGenerator`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-a</code></td>
<td>The generator <code>a</code> is executed depending on the availability of an attribute. The optional generator succeeds if its embedded generator succeeds (unless the underlying output stream reports an error).</td>
</tr>
</tbody>
</table>

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-a (optional, unary -)</code></td>
<td><code>a: A --&gt; -a: optional&lt;A&gt;</code>&lt;br&gt;<code>a: Unused --&gt; -a: Unused</code></td>
</tr>
</tbody>
</table>

**Important**

The table above uses `optional<A>` as a placeholder only. The notation of `optional<A>` stands for the data type `boost::optional<A>`. The optional generator will execute its embedded generator once if the provided attribute holds a valid value. It forwards the value held in its attribute to the embedded generator.

It is important to note, that the optional generator does not perform any buffering of the output generated by its embedded elements. That means that any failing element might have already generated some output, which is not rolled back.

**Tip**

The simplest way to force a optional generator to behave as if it did buffering is to wrap it into a buffering directive (see `buffer`):

```cpp
buffer[-a]
```

which will not generate any output in case of a failing generator `-a`. 
Complexity

The overall complexity of the optional generator is defined by the complexity of its embedded generator. The complexity of the optional generator itself is $O(1)$.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::double_;  
```

Basic usage of an optional generator:

```cpp
boost::optional<double> val(1.0);
    test_generator_attr("1.0", -double_, val);
    test_generator_attr("2.0", -double_, 2.0);
```

Usage and result of an empty optional generator:

```cpp
boost::optional<double> val;       // empty optional
    test_generator_attr("", -double_, val);
```

And-Predicate Generator ($\&a$)

Description

The and-predicate generator is used to test, whether the embedded generator succeeds without generating any output. It succeeds if the embedded generator succeeds.

Header

```cpp
// forwards to <boost/spirit/home/karma/operator/and_predicate.hpp>
#include <boost/spirit/include/karma_and_predicate.hpp>
```

Also, see Include Structure.

Model of

`UnaryGenerator`
Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in UnaryGenerator.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&amp; a$</td>
<td>The generator $a$ is executed for the sole purpose of testing whether it succeeds. The and-predicate generator succeeds if its embedded generator succeeds (unless the underlying output stream reports an error). The and-predicate never produces any output.</td>
</tr>
</tbody>
</table>

The and generator is implemented by redirecting all output produced by its embedded generator into a discarding device.

Attributes

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&amp; a$ (and-predicate, unary $&amp;$)</td>
<td>$a: A \rightarrow &amp; a: A$</td>
</tr>
</tbody>
</table>

Note

The attribute of the and-predicate is not always unused_type, which is different from Qi's and-predicate. This is necessary as the generator the and predicate is attached to most of the time needs an attribute.

Complexity

The overall complexity of the and-predicate generator is defined by the complexity of its embedded generator. The complexity of the and-predicate generator itself is $O(1)$.

Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <iostream>
#include <string>
```

Some using declarations:
using boost::spirit::karma::double_;  
using boost::spirit::karma::ascii::char_;  
using boost::spirit::karma::ascii::string;  
using boost::phoenix::ref;

Basic usage of an and predicate generator:

```cpp
test_generator_attr("b", &char_("a") << 'b' | 'c', 'a');
test_generator_attr("c", &char_("a") << 'b' | 'c', 'x');

test_generator_attr("abc", &string("123") << "abc" | "def", "123");
test_generator_attr("def", &string("123") << "abc" | "def", "456");
```

Not-Predicate Generator (\(!a\))

**Description**

The not-predicate generator is used to test, whether the embedded generator fails, without generating any output. It succeeds if the embedded generator fails.

**Header**

```cpp
// forwards to <boost/spirit/home/karma/operator/not_predicate.hpp>
#include <boost/spirit/include/karma_not_predicate.hpp>
```

Also, see Include Structure.

**Model of**

UnaryGenerator

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in UnaryGenerator.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>!a</td>
<td>The generator (a) is executed for the sole purpose of testing whether it succeeds. The not-predicate generator succeeds if its embedded generator fails (unless the underlying output stream reports an error). The not-predicate never produces any output.</td>
</tr>
</tbody>
</table>

The not generator is implemented by redirecting all output produced by its embedded generator into a discarding device.

**Attributes**

See Compound Attribute Notation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>!a (not-predicate, unary !)</td>
<td>a: A --&gt; !a: A</td>
</tr>
</tbody>
</table>
Note
The attribute of the not-predicate is not always unused_type, which is different from Qi's not-predicate. This is necessary as the generator the and-predicate is attached to most of the time needs an attribute.

Complexity
The overall complexity of the not-predicate generator is defined by the complexity of its embedded generator. The complexity of the not-predicate generator itself is O(1).

Example

Note
The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::double_;
using boost::spirit::karma::ascii::char_;
using boost::spirit::karma::ascii::string;
using boost::phoenix::ref;
```

Basic usage of a not predicate generator:

```cpp
test_generator_attr("c", !char_('a') << 'b' | 'c', 'a');
test_generator_attr("b", !char_('a') << 'b' | 'c', 'x');
test_generator_attr("def", !string("123") << "abc" | "def", "123");
test_generator_attr("abc", !string("123") << "abc" | "def", "456");
```

Stream Generators

This module includes the description of the different variants of the stream generator. It can be used to utilize existing streaming operators (operator<<(std::ostream&, ...)) for output generation.

Header

```cpp
// forwards to <boost/spirit/home/karma/stream.hpp>
#include <boost/spirit/include/karma/stream.hpp>
```

Also, see Include Structure.
Stream Generators (stream, wstream, etc.)

Description

The `stream_generator` is a primitive which allows to use pre-existing standard streaming operators for output generation integrated with Spirit.Karma. It provides a wrapper generator dispatching the value to output to the stream operator of the corresponding type. Any value to be formatted using the `stream_generator` will result in invoking the standard streaming operator for its type `A`, for instance:

```cpp
std::ostream& operator<<(std::ostream&, A const&);
```

Header

```cpp
#include <boost/spirit/include/karma_stream.hpp>
```

Also, see Include Structure.

Namespace

```cpp
namespace boost::spirit::stream // alias: boost::spirit::karma::stream
namespace boost::spirit::wstream // alias: boost::spirit::karma::wstream
```

Synopsis

```cpp
template <typename Char>
struct stream_generator;
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char</td>
<td>The character type to use to generate the output. This type will be used while assigning the generated characters to the underlying output iterator.</td>
<td>char</td>
</tr>
</tbody>
</table>

Model of

`PrimitiveGenerator`

Notation

- A variable instance of any type with a defined matching streaming `operator<<()` or a Lazy Argument that evaluates to any type with a defined matching streaming `operator<<()`.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in `PrimitiveGenerator`. 
Call the streaming operator\texttt{<<()} for the type of the mandatory attribute. The output emitted by this operator will be the result of the stream generator. This generator never fails (unless the underlying output stream reports an error). The character type of the I/O ostringstream is assumed to be \texttt{char}.

Call the streaming operator\texttt{<<()} for the type of the immediate value \(s\). The output emitted by this operator will be the result of the stream generator. This generator never fails (unless the underlying output stream reports an error). The character type of the I/O ostringstream is assumed to be \texttt{char}.

Call the streaming operator\texttt{<<()} for the type of the mandatory attribute. The output emitted by this operator will be the result of the stream generator. This generator never fails (unless the underlying output stream reports an error). The character type of the I/O ostringstream is assumed to be \texttt{wchar_t}.

Call the streaming operator\texttt{<<()} for the type of the immediate value \(s\). The output emitted by this operator will be the result of the stream generator. This generator never fails (unless the underlying output stream reports an error). The character type of the I/O ostringstream is assumed to be \texttt{wchar_t}.

All generators listed in the table above are predefined specializations of the \texttt{stream\_generator<Char>} basic stream generator type described below. It is possible to directly use this type to create stream generators using an arbitrary underlying character type.

All of the stream generators listed above require the type of the value to generate output for (either the immediate value or the associated attribute) to implement a streaming operator conforming to the usual I/O streams conventions (where \texttt{attribute\_type} is the type of the value to generate output for):
template <typename Ostream>
Ostream& operator<<(Ostream& os, attribute_type const& attr)
{
    // type specific output generation
    return os;
}

This operator will be called by the stream generators to gather the output for the attribute of type attribute_type. All data streamed into the given Ostream will end up being generated by the corresponding stream generator instance.

**Note**

If the stream generator is invoked inside a format (or format_delimited) stream manipulator the Ostream passed to the operator<<() will have registered (imbued) the same standard locale instance as the stream the format (or format_delimited) manipulator has been used with. This ensures all facets registered (imbued) with the original I/O stream object are used during output generation.

### Attributes

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>hold_any, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>stream(s)</td>
<td>unused</td>
</tr>
<tr>
<td>wstream</td>
<td>whold_any, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>wstream(s)</td>
<td>unused</td>
</tr>
<tr>
<td>stream_generator&lt;Char&gt;()</td>
<td>basic_hold_any&lt;Char&gt;, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td>stream_generator&lt;Char&gt;()(s)</td>
<td>unused</td>
</tr>
</tbody>
</table>

**Important**

The attribute type hold_any exposed by some of the stream generators is semantically and syntactically equivalent to the type implemented by Boost.Any. It has been added to Spirit as it has better a performance and a smaller footprint if compared to Boost.Any.

**Note**

In addition to their usual attribute of type Attrib all listed generators accept an instance of a boost::optional<Attrib> as well. If the boost::optional<> is initialized (holds a value) the generators behave as if their attribute was an instance of Attrib and emit the value stored in the boost::optional<>. Otherwise the generators will fail.

### Complexity

O(N), where N is the number of characters emitted by the stream generator
Example

Note

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::stream;
```

And a class definition used in the examples:

```cpp
// a simple complex number representation z = a + bi
struct complex
{
    complex (double a, double b)
    : a(a), b(b)
    {}

    double a;
    double b;
};
```

```cpp
// define streaming operator for the type complex
std::ostream&
operator<< (std::ostream& os, complex const& z)
{
    os << "{ " << z.a << "," << z.b << "}";
    return os;
}
```

Basic usage of stream generators:

```cpp
test_generator_attr("abc", stream, "abc");
test_generator("abc", stream("abc"));
test_generator_attr("{1.2,2.4}", stream, complex(1.2, 2.4));
test_generator("{1.2,2.4}" , stream(complex(1.2, 2.4)));
```

String Generators

This module includes different string oriented generators allowing to output character sequences. It includes the symbols generator and variants of the string generator.
Module Header

```cpp
// forwards to <boost/spirit/home/karma/string.hpp>
#include <boost/spirit/include/karma_string.hpp>
```

Also, see Include Structure.

### String Generators ((string, lit)

#### Description

The string generators described in this section are:

The `string` generator emits a string of characters. The `string` generator is implicitly verbatim: the delimit parser is not applied in between characters of the string. The `string` generator has an associated Character Encoding Namespace. This is needed when doing basic operations such as forcing lower or upper case. Examples:

```cpp
text("Hello")
text(L"Hello")
text(s)    // s is a std::string
```

`lit`, like `string`, also emits a string of characters. The main difference is that `lit` does not consumes an attribute. A plain string like "hello" or a std::basic_string is equivalent to a `lit`. Examples:

```cpp
"Hello"
lit("Hello")
lit(L"Hello")
lit(s)    // s is a std::string
```

#### Header

```cpp
// forwards to <boost/spirit/home/karma/string/lit.hpp>
#include <boost/spirit/include/karma_string.hpp>
```

Also, see Include Structure.

#### Namespace

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::lit // alias: boost::spirit::karma::lit</td>
<td>ns::string</td>
</tr>
</tbody>
</table>

In the table above, \textit{ns} represents a Character Encoding Namespace used by the corresponding string generator.

#### Model of

- **PrimitiveGenerator**

#### Notation

- A Character-class specific string (See Character Class Types), or a Lazy Argument that evaluates to a character-class specific string value
- The type of a character-class specific string `s`. 
A Character Encoding Namespace.

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in `PrimitiveGenerator`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s</code></td>
<td>Generate the string literal <code>s</code>. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>lit(s)</code></td>
<td>Generate the string literal <code>s</code>. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>ns::string</code></td>
<td>Generate the string provided by a mandatory attribute interpreted in the character set defined by <code>ns</code>. This generator never fails (unless the underlying output stream reports an error).</td>
</tr>
<tr>
<td><code>ns::string(s)</code></td>
<td>Generate the string <code>s</code> as provided by the immediate literal value the generator is initialized from. If this generator has an associated attribute it succeeds only if the attribute is equal to the immediate literal (unless the underlying output stream reports an error). Otherwise this generator fails and does not generate any output.</td>
</tr>
</tbody>
</table>

**Note**

The generators `lit(s)` and `string(s)` can be initialized either using a string literal value (i.e. "abc"), or using a `std::basic_string<char_type, ...>`, where `char_type` is the required value type of the underlying character sequence.

**Caution**

The generator `string(s)` up to version 2.4.1 of Spirit has an undocumented feature. Given argument `s` generator succeeds as long as `s` is a prefix of given attribute. This problem has been fixed in Spirit V2.4.2.

**Attributes**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s</code></td>
<td>unused</td>
</tr>
<tr>
<td><code>lit(s)</code></td>
<td>unused</td>
</tr>
<tr>
<td><code>ns::string</code></td>
<td><code>s</code>, attribute is mandatory (otherwise compilation will fail)</td>
</tr>
<tr>
<td><code>ns::string(s)</code></td>
<td><code>s</code>, attribute is optional, if it is supplied, the generator compares the attribute with <code>s</code> and succeeds only if both are equal, failing otherwise</td>
</tr>
</tbody>
</table>
In addition to their usual attribute of type \texttt{S} all listed generators accept an instance of a \texttt{boost::optional\textless S\textgreater} as well. If the \texttt{boost::optional<>} is initialized (holds a value) the generators behave as if their attribute was an instance of \texttt{S} and emit the value stored in the \texttt{boost::optional<>}. Otherwise the generators will fail.

**Complexity**

O(N), where N is the number of characters emitted by the string generator

**Example**

The test harness for the example(s) below is presented in the Basics Examples section.

Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::lit;
using boost::spirit::ascii::string;
```

Basic usage of string generators:

```cpp
test_generator("abc", "abc");
test_generator("abc", lit("abc"));
test_generator("abc", lit(std::string("abc")));

test_generator_attr("abc", string, "abc");
test_generator("abc", string("abc"));
test_generator("abc", string(std::string("abc")));

test_generator_attr("abc", string("abc"), "abc");
test_generator_attr("", string("abc"), "cba"); // fails (as "abc" != "cba")
```

**Symbols Generator (symbols)**

**Description**

The class \texttt{symbols} implements an 'inverse' symbol table: an associative container (or map) of key-value pairs where the values are (most of the time) strings. It maps the value to be generated (the key) to any other value which will be emitted instead of the original key.

The Karma symbol table class \texttt{symbols} is a generator, an instance of which may be used anywhere in the grammar specification. It is an example of a dynamic generator. A dynamic generator is characterized by its ability to modify its behavior at run time. Initially, an empty symbols object will emit nothing. At any time, symbols may be added, thus, dynamically altering its behavior.
Header

// forwards to <boost/spirit/home/karma/string/symbols.hpp>
#include <boost/spirit/include/karma_symbols.hpp>

Also, see Include Structure.

Namespace

Name
boost::spirit::karma::symbols

Synopsis

template <typename Attrib, typename T, typename Lookup,
        typename CharEncoding, typename Tag>
struct symbols;

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attrib</td>
<td>The type of the original attribute to be used as the key into the symbol generator (the symbol).</td>
<td>char</td>
</tr>
<tr>
<td>T</td>
<td>The data type associated with each key.</td>
<td>unused_type</td>
</tr>
<tr>
<td>Lookup</td>
<td>The symbol search implementation</td>
<td>if T is unused_type, std::set&lt;Attrib&gt;, and std::map&lt;Attrib, T&gt; otherwise</td>
</tr>
<tr>
<td>CharEncoding</td>
<td>Used for character set selection, normally not used by end user.</td>
<td>unused_type</td>
</tr>
<tr>
<td>Tag</td>
<td>Used for character set selection, normally not used by end user.</td>
<td>unused_type</td>
</tr>
</tbody>
</table>

Model of

PrimitiveGenerator

Notation

Sym A symbols type.
Attrib An attribute type.
T A data type.
sym, sym2 symbols objects.
sseq An STL container of strings.
dseq An STL container of data with value_type T.
A String,

Objects of type $T$.

A callable function or function object.

ForwardIterator first/last pair.

**Expression Semantics**

Semantics of an expression is defined only where it differs from, or is not defined in *PrimitiveGenerator*. 
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sym()</td>
<td>Construct an empty symbols object instance named &quot;symbols&quot;.</td>
</tr>
<tr>
<td>Sym(name)</td>
<td>Construct an empty symbols object instance named name.</td>
</tr>
<tr>
<td>Sym(sym2)</td>
<td>Copy construct a symbols from sym2 (Another symbols object).</td>
</tr>
<tr>
<td>Sym(sseq)</td>
<td>Construct symbols from sseq (An STL container of symbols of type Attrib) named &quot;symbols&quot;.</td>
</tr>
<tr>
<td>Sym(sseq, name)</td>
<td>Construct symbols from sseq (an STL container of symbols of type Attrib) named name.</td>
</tr>
<tr>
<td>Sym(sseq, dseq)</td>
<td>Construct symbols from sseq and dseq (An STL container of symbols of type Attrib and an STL container of data with value_type T) which is named &quot;symbols&quot;.</td>
</tr>
<tr>
<td>Sym(sseq, dseq, name)</td>
<td>Construct symbols from sseq and dseq (An STL container of symbols of type Attrib and an STL container of data with value_type T) which is named name.</td>
</tr>
<tr>
<td>sym = sym2</td>
<td>Assign sym2 to sym.</td>
</tr>
<tr>
<td>sym = s1, s2, ..., sN</td>
<td>Assign one or more symbols (s1...sN) to sym. The associated data values of type T are default constructed.</td>
</tr>
<tr>
<td>sym += s1, s2, ..., sN</td>
<td>Add one or more symbols (s1...sN) to sym. The associated data values of type T are default constructed.</td>
</tr>
<tr>
<td>sym.add(s1)(s2)...(sN)</td>
<td>Add one or more symbols (s1...sN) to sym. The associated data values of type T are default constructed.</td>
</tr>
<tr>
<td>sym.add(s1, d1)(s2, d2)...(sN, dN)</td>
<td>Add one or more symbols (s1...sN) with associated data (d1...dN) to sym.</td>
</tr>
<tr>
<td>sym -= s1, s2, ..., sN</td>
<td>Remove one or more symbols (s1...sN) from sym.</td>
</tr>
<tr>
<td>sym.remove(s1)(s2)...(sN)</td>
<td>Remove one or more symbols (s1...sN) from sym.</td>
</tr>
<tr>
<td>sym.clear()</td>
<td>Erase all of the symbols in sym.</td>
</tr>
<tr>
<td>sym.at(s)</td>
<td>Return a reference to the object associated with symbol, s. If sym does not already contain such an object, at inserts the default object T().</td>
</tr>
<tr>
<td>sym.find(s)</td>
<td>Return a pointer to the object associated with symbol, s. If sym does not already contain such an object, find returns a null pointer.</td>
</tr>
<tr>
<td>sym.for_each(f)</td>
<td>For each symbol in sym s invoke f(typename Lookup::value_type).</td>
</tr>
<tr>
<td>sym.name()</td>
<td>Retrieve the current name of the symbols object.</td>
</tr>
<tr>
<td>sym.name(name)</td>
<td>Set the current name of the symbols object to be name.</td>
</tr>
</tbody>
</table>
The symbols generator uses the supplied attribute as the key to be looked up in the internal associative container. If the key exists the generator emits the associated value and succeeds (unless the underlying output stream reports an error). If the value type stored in the symbol generator is `unused_type` it will emit the key instead. If the key does not exist the generator fails while not emitting anything.

**Attributes**

The attribute of `symbol<Attrib, T>` is `Attrib`.

If the supplied attribute is a `Boost.Fusion` sequence, then the symbol table generator will use the first element of that `Boost.Fusion` sequence as the key to be used for lookup. The type of that first element needs to be convertible to `Attrib`. In this case the second element of the `Boost.Fusion` sequence is used as the attribute while calling a generator derived from the value stored in the symbol table for the found entry.

If the supplied attribute is a container type (`traits::is_container` resolves to `mpl::true_`), then the symbol table generator will use the first element stored in that container as the key to be used for lookup. The `value_type` (returned by `traits::container_value`) has to be convertible to `Attrib`. In this case the second element stored in that container is used as the attribute while calling a generator derived from the value stored in the symbol table for the found entry.

If the supplied attribute is not a `Boost.Fusion` sequence and not a container type, the supplied attribute is directly used as the key for item lookup. The attribute is used as the attribute while calling a generator derived from the value stored in the symbol table for the found entry.

In any case, because the supplied key (i.e. either the first element of the `Boost.Fusion` sequence, the first container element, or the attribute otherwise) is passed as the attribute to a generator derived from the value stored in the symbol table for the found entry, the symbol table may store generators, which will produce output based on that value. For instance:

```cpp
// The symbol table maps a single character key to a rule<>
// The rule<> exposes an attribute of char as well
rule<output_iterator_type, char()> r1 = char_;

symbols<char, rule<output_iterator_type, char()> > sym;
sym.add
  ('j', r1.alias())
  ('h', r1.alias())
  ('t', r1.alias())
  ('k', r1.alias())
;
// Supplying a fusion vector as the attribute will use the first element
// (the 'j') as the key to be looked up, while the second element (the 'J')
// is passed on as the attribute to the rule<> stored in the symbol table.
// Consequently, the example generates a single 'J'.
BOOST_ASSERT(test("J", sym, make_vector('j', 'J')));
```

**Complexity**

The default implementation uses a `std::map<>` or a `std::set<>` with a complexity of:

\[ O(\log n) \]

Where \( n \) is the number of stored symbols.

**Example**

```
Note

The test harness for the example(s) below is presented in the Basics Examples section.
```
Some includes:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <boost/spirit/include/support_utree.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/fusion/include/std_pair.hpp>
#include <iostream>
#include <string>
```

Some using declarations:

```cpp
using boost::spirit::karma::symbols;
```

Basic usage of symbol generators:

```cpp
symbols<char, char const*> sym;

sym.add
  ('a', "Apple")
  ('b', "Banana")
  ('o', "Orange")
;

test_generator_attr("Banana", sym, 'b');
```

## Performance Measurements

### Performance of Numeric Generators

#### Comparing the performance of a single int_generator

These performance measurements are centered around default formatting of a single `int` integer number using different libraries and methods. The overall execution times for those examples are compared below. We compare using `sprintf`, C++ iostreams, `Boost.Format`, and `Spirit.Karma`.

For the full source code of the performance test please see here: `int_generator.cpp`. All the measurements have been done by executing $1e7$ iterations for each formatting type (`NUMITERATIONS` is set to $1e7$ in the code shown below).

Code used to measure the performance for `ltoa`:

```cpp
char buffer[65]; // we don't expect more than 64 bytes to be generated here
for (int i = 0; i < MAX_ITERATION; ++i)
{
    ltoa(v[i], buffer, 10);
}
```

Code used to measure the performance for standard C++ iostreams:
Code used to measure the performance for **Boost.Format**:

```cpp
std::stringstream str;
for (int i = 0; i < MAX_ITERATION; ++i)
{
    str.str("\n");
    str << v[i];
}
```

Code used to measure the performance for **Spirit.Karma** using a plain character buffer:

```cpp
char buffer[65]; // we don't expect more than 64 bytes to be generated here
for (int i = 0; i < MAX_ITERATION; ++i)
{
    char *ptr = buffer;
    karma::generate(ptr, int_, v[i]);
    *ptr = '\0';
}
```

The following table shows the overall performance results collected while using different compilers. All times are in seconds measured for `1e7` iterations (platform: Windows7, Intel Core Duo(tm) Processor, 2.8GHz, 4GByte RAM). For a more readable comparison of the results see this [figure](https://example.com).

**Table 5. Performance comparison for a single int (all times in [s], `1e7` iterations)**

<table>
<thead>
<tr>
<th>Library</th>
<th>gcc 4.4.0 (32 bit)</th>
<th>VC++ 10 (32 bit)</th>
<th>Intel 11.1 (32 bit)</th>
<th>gcc 4.4.0 (64 bit)</th>
<th>VC++ 10 (64 bit)</th>
<th>Intel 11.1 (64 bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>itoa</strong></td>
<td>1.542</td>
<td>0.895</td>
<td>0.884</td>
<td>1.163</td>
<td>1.099</td>
<td>0.906</td>
</tr>
<tr>
<td><strong>Spirit.Karma</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>int_</strong></td>
<td>1.421</td>
<td>0.744</td>
<td>0.697</td>
<td>1.072</td>
<td>0.953</td>
<td>0.606</td>
</tr>
</tbody>
</table>

369
Figure 3. Performance comparison for a single int

![Performance Comparison Graph]

Comparing the performance of a single double generator

These performance measurements are centered around default formatting of a single double floating point number using different libraries and methods. The overall execution times for those examples are compared below. We compare using `sprintf`, C++ `iostreams`, `Boost.Format`, and `Spirit.Karma`.

For the full source code of the performance test please see here: `double_performance.cpp`. All the measurements have been done by executing $1e6$ iterations for each formatting type (NUMITERATIONS is set to $1e6$ in the code shown below).

Code used to measure the performance for `sprintf`:

```c
char buffer[256];
for (int i = 0; i < NUMITERATIONS; ++i) {
    sprintf(buffer, "%f", 12345.12345);
}
```

Code used to measure the performance for standard C++ `iostreams`:

```c
stringstream strm;
for (int i = 0; i < NUMITERATIONS; ++i) {
    strm.str(""');
    strm << 12345.12345;
}
```

Code used to measure the performance for `Boost.Format`:

---

370
The following code shows the common definitions used by all *Spirit.Karma* performance measurements as listed below:

```cpp
std::string generated;
boost::format double_format("%f");
for (int i = 0; i < NUMITERATIONS; ++i)
    generated = boost::str(double_format % 12345.12345);
```

Code used to measure the performance for *Spirit.Karma* using a plain character buffer:

```cpp
char buffer[256];
for (int i = 0; i < NUMITERATIONS; ++i) {
    char *p = buffer;
    generate(p, double_, 12345.12345);
    *p = '\0';
}
```

The following table shows the overall performance results collected while using different compilers. All times are in seconds measured for $1e6$ iterations (platform: Windows7, Intel Core Duo(tm) Processor, 2.8GHz, 4GByte RAM). For a more readable comparison of the results see this figure.

### Table 6. Performance comparison for a single double (all times in [s], `$1e6$` iterations)

<table>
<thead>
<tr>
<th>Library</th>
<th>gcc 4.4.0 (32 bit)</th>
<th>VC++ 10 (32 bit)</th>
<th>Intel 11.1 (32 bit)</th>
<th>gcc 4.4.0 (64 bit)</th>
<th>VC++ 10 (64 bit)</th>
<th>Intel 11.1 (64 bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sprintf</strong></td>
<td>0.755</td>
<td>0.965</td>
<td>0.880</td>
<td>0.713</td>
<td>0.807</td>
<td>0.694</td>
</tr>
<tr>
<td><strong>iostreams</strong></td>
<td>2.316</td>
<td>2.624</td>
<td>1.964</td>
<td>1.634</td>
<td>1.468</td>
<td>1.354</td>
</tr>
<tr>
<td><strong>Boost.Format</strong></td>
<td>3.188</td>
<td>3.737</td>
<td>2.878</td>
<td>3.217</td>
<td>2.672</td>
<td>2.011</td>
</tr>
<tr>
<td><strong>Spirit.Karma double_</strong></td>
<td>0.813</td>
<td>0.561</td>
<td>0.368</td>
<td>0.426</td>
<td>0.260</td>
<td>0.218</td>
</tr>
</tbody>
</table>
Comparing the performance of a sequence of several generators

These performance measurements are centered around formatting of a sequence of different items, including double floating point numbers using different libraries and methods. The overall execution times for those examples are compared below. We compare using sprintf, C++ iostreams, Boost.Format, and Spirit.Karma.

For the full source code of the performance test please see here: format_performance.cpp. All the measurements have been done by doing \(1 \times 10^6\) iterations for each formatting type (NUMITERATIONS is set to \(1 \times 10^6\)).

Code used to measure the performance for sprintf:

```c
char buffer[256];
for (int i = 0; i < NUMITERATIONS; ++i) {
    sprintf(buffer, "[%-14.3f%-14.3f]", 12345.12345, 12345.12345);
}
```

Code used to measure the performance for standard iostreams:
std::stringstream strm;
for (int i = 0; i < NUMITERATIONS; ++i) {
    strm.str("\n");
    strm << '['
        << std::setiosflags(std::ios::fixed)
        << std::left
        << std::setprecision(3)
        << std::setw(14)
        << 12345.12345
        << std::setw(14)
        << 12345.12345
        << ']
;}

Code used to measure the performance for Boost.Format:

std::string generated;
boost::format outformat("[%-14.3f%-14.3f]");
for (int i = 0; i < NUMITERATIONS; ++i)
    generated = boost::str(outformat % 12345.12345 % 12345.12345);

The following code shows the common definitions used by all Spirit.Karma performance measurements as listed below:

```
template <typename T>
struct double3_policy : boost::spirit::karma::real_policies<T>
{
    // we want to generate up to 3 fractional digits
    static unsigned int precision(T) { return 3; }
};

typedef boost::spirit::karma::real_generator<double, double3_policy<double> >
    double3_type;

double3_type const double3 = double3_type();
```

Code used to measure the performance for Spirit.Karma using a plain character buffer:

```
char buffer[256];
for (int i = 0; i < NUMITERATIONS; ++i) {
    char *p = buffer;
generate(p
        , '[' << left_align(14)[double3] << left_align(14)[double3] << ']
        , 12345.12345, 12345.12345);
    *p = '\0';
}
```

The following table shows the overall performance results collected while using different compilers. All times are in seconds measured for 1e6 iterations (platform: Windows7, Intel Core Duo(tm) Processor, 2.8GHz, 4GByte RAM). For a more readable comparison of the results see this figure.
Table 7. Performance comparison for a sequence of several items (all times in [s], `1e6` iterations)

<table>
<thead>
<tr>
<th>Library</th>
<th>gcc 4.4.0 (32 bit)</th>
<th>VC++ 10 (32 bit)</th>
<th>Intel 11.1 (32 bit)</th>
<th>gcc 4.4.0 (64 bit)</th>
<th>VC++ 10 (64 bit)</th>
<th>Intel 11.1 (64 bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sprintf</td>
<td>1.725</td>
<td>1.892</td>
<td>1.903</td>
<td>1.469</td>
<td>1.608</td>
<td>1.493</td>
</tr>
<tr>
<td>iostreams</td>
<td>4.827</td>
<td>5.287</td>
<td>4.444</td>
<td>3.112</td>
<td>3.319</td>
<td>2.877</td>
</tr>
<tr>
<td>Boost.Format</td>
<td>5.881</td>
<td>7.089</td>
<td>5.801</td>
<td>5.455</td>
<td>5.254</td>
<td>4.164</td>
</tr>
<tr>
<td>Spirit.Karma</td>
<td>1.942</td>
<td>1.242</td>
<td>0.999</td>
<td>1.334</td>
<td>0.758</td>
<td>0.686</td>
</tr>
</tbody>
</table>

Figure 5. Performance comparison for a sequence of several items

Karma Index

Symbols

_1 ... _N
Phoenix, 80, 235
_a ... _j
Phoenix, 80, 235
_pass
Phoenix, 80, 235
_r1 ... _rN
Phoenix, 80, 235
_val
Phoenix, 80, 235
A
alnum
Character Classification Generators (alnum, digit, etc.), 280
alpha
Character Classification Generators (alnum, digit, etc.), 280
alternative (a | b)
   Alternative Generator (a | b), 343
and-predicate (&a)
   And-Predicate Generator (& a), 353
as<T
   Generator Directives Forcing Atomic Extraction (as<T>, as_string[], as_wstring[]), 308
as_string
   Generator Directives Forcing Atomic Extraction (as<T>, as_string[], as_wstring[]), 308
as_wstring
   Generator Directives Forcing Atomic Extraction (as<T>, as_string[], as_wstring[]), 308
attr_cast
   Attribute Transformation Pseudo Generator (attr_cast), 259
auto
   Auto Generator, 255

B
big_dword
   Binary Big Endianness Generators, 273
big_qword
   Binary Big Endianness Generators, 273
big_word
   Binary Big Endianness Generators, 273
bin
   Unsigned Integer Number Generators (uint_, etc.), 316
blank
   Character Classification Generators (alnum, digit, etc.), 280
bool
   Boolean Generators (bool_), 336
bool_generator
   Boolean Generators (bool_), 336
bool_policies
   Boolean Generators (bool_), 336
buffer
   Generator Directive for Temporary Output Buffering (buffer[]), 300
repetition
generator
directive (repeat[]), 289
byte_
   Binary Native Endianness Generators, 267

c
center
   Alignment Generator Directives (left_align[], center[], right_align[]), 284
char
   Character Generators (char_, lit), 275
cntrl
   Character Classification Generators (alnum, digit, etc.), 280
columns
   Alignment Generator Directives (left_align[], center[], right_align[]), 284
generator
directive separating output into columns (columns[]), 306
create_generator
   API for Automatic Generator Creation, 251
create_generator_exists
   API for Automatic Generator Creation, 251
D

delimit
   Generator Directives Controlling Automatic Delimiting (verbatim [], no_delimit [], delimit []), 293
delimit_flag::dont_delimit
   Iterator Based Generator API, 245
   Stream Based Generator API, 248
delimit_flag::pre_delimit
   Iterator Based Generator API, 245
   Stream Based Generator API, 248
digit
   Character Classification Generators (alnum, digit, etc.), 280
double_
   Real Number Generators (float_, double_, etc.), 326
duplicate
   Generator Directive Duplicating Attributes (duplicate []), 304
dword
   Binary Native Endianness Generators, 267

E
eol
   End of Line Generator (eol), 261
eps
   Epsilon Generator (eps), 262

F
false_
   Boolean Generators (bool_), 336
float_
   Real Number Generators (float_, double_, etc.), 326
format
   Stream Based Generator API, 248
format_delimited
   Stream Based Generator API, 248

G
generate
   Iterator Based Generator API, 245
generate_delimited
   Iterator Based Generator API, 245
generator
   BinaryGenerator, 239
   Generator, 236
   Generator Concepts, 235
   NaryGenerator, 240
   Nonterminal, 240
   PrimitiveGenerator, 237
   UnaryGenerator, 237, 238
grammar
   Generator Grammar, 313, 314
graph
   Character Classification Generators (alnum, digit, etc.), 280

H
hex
   Unsigned Integer Number Generators (uint_, etc.), 316
I
inf
   Repetition Generator Directive ( repeat [] ), 289, 290
int_
   Signed Integer Number Generators ( int_ , etc.), 321
int_generator
   Signed Integer Number Generators ( int_ , etc.), 321

K
kleene (*a)
   Kleene Star Generator ( * a ), 345

L
lazy
   Lazy Generator ( lazy ), 265
left_align
   Alignment Generator Directives ( left_align [], center [], right_align [] ), 284
list (a % b)
   List Generator ( a % b ), 349
lit
   Boolean Generators ( bool_ ), 336
   Character Generators ( char_ , lit ), 275
   Real Number Generators ( float_ , double_ , etc.), 326
   Signed Integer Number Generators ( int_ , etc.), 321
   String Generators ( string , lit ), 361
   Unsigned Integer Number Generators ( uint_ , etc.), 316
little_dword
   Binary Little Endianness Generators, 270
little_qword
   Binary Little Endianness Generators, 270
little_word
   Binary Little Endianness Generators, 270
long_
   Signed Integer Number Generators ( int_ , etc.), 321
long_double
   Real Number Generators ( float_ , double_ , etc.), 326
long_long
   Signed Integer Number Generators ( int_ , etc.), 321
lower
   Character Classification Generators ( alnum , digit , etc.), 280
   Generator Directives Controlling Case Sensitivity ( upper [], lower [] ), 296
   Repetition Generator Directive ( repeat [] ), 289

M
maxwidth
   Alignment Generator Directives ( left_align [], center [], right_align [] ), 284
   Generator Directives Controlling the Maximum Field Width ( maxwidth [] ), 298

N
not-predicate (!a)
   Not-Predicate Generator ( ! a ), 355
no_delimit
   Generator Directives Controlling Automatic Delimiting ( verbatim [], no_delimit [], delimit [] ), 293

O
oct
Unsigned Integer Number Generators (uint_, etc.), 316

omit

Generator Directives Consuming Attributes (omit [] and skip []), 302

optional (-a)

Optional Generator (-a), 351

P

placeholders

Phoenix, 80, 235

plus (+a)

Plus Generator (+a), 347

print

Character Classification Generators (alnum, digit, etc.), 280

punct

Character Classification Generators (alnum, digit, etc.), 280

Q

qword

Binary Native Endianness Generators, 267

R

real_generator

Real Number Generators (float_, double_, etc.), 326

real_policies

Real Number Generators (float_, double_, etc.), 326

repeat

Repetition Generator Directive (repeat []), 289, 290

right_align

Alignment Generator Directives (left_align [], center [], right_align []), 284

rule

Generator Rule, 310, 311

S

Semantic Action

Semantic Actions with Generators, 253

sequence (a << b)

Sequence Generator (a << b), 341

short_

Signed Integer Number Generators (int_, etc.), 321

space

Character Classification Generators (alnum, digit, etc.), 280

string

String Generators (string, lit), 361

Symbols Generator (symbols), 363, 365

symbols

Symbols Generator (symbols), 363, 364

T

ture_

Boolean Generators (bool_), 336

U

uint_

Unsigned Integer Number Generators (uint_, etc.), 316

uint_generator

Signed Integer Number Generators (int_, etc.), 321
Unsigned Integer Number Generators (uint_, etc.), 316
ulong_
  Unsigned Integer Number Generators (uint_, etc.), 316
ulong_long
  Unsigned Integer Number Generators (uint_, etc.), 316
upper
  Character Classification Generators (alnum, digit, etc.), 280
  Generator Directives Controlling Case Sensitivity (upper[], lower[]), 296
  Repetition Generator Directive (repeat[]), 289, 290
ushort_
  Unsigned Integer Number Generators (uint_, etc.), 316

V
verbatim
  Generator Directives Controlling Automatic Delimiting (verbatim[], no_delimit[], delimit[]), 293

W
word
  Binary Native Endianness Generators, 267

X
xdigit
  Character Classification Generators (alnum, digit, etc.), 280
Lex - Writing Lexical Analyzers

Introduction to Spirit.Lex

Lexical scanning is the process of analyzing the stream of input characters and separating it into strings called tokens, separated by whitespace. Most compiler texts start here, and devote several chapters to discussing various ways to build scanners. Spirit.Lex is a library built to take care of the complexities of creating a lexer for your grammar (in this documentation we will use the terms 'lexical analyzer', 'lexer' and 'scanner' interchangeably). All that is needed to create a lexer is to know the set of patterns describing the different tokens you want to recognize in the input. To make this a bit more formal, here are some definitions:

• A token is a sequence of consecutive characters having a collective meaning. Tokens may have attributes specific to the token type, carrying additional information about the matched character sequence.

• A pattern is a rule expressed as a regular expression and describing how a particular token can be formed. For example, \[A-Za-z]\ [A-Za-z_0-9]* is a pattern for a rule matching C++ identifiers.

• Characters between tokens are called whitespace; these include spaces, tabs, newlines, and formfeeds. Many people also count comments as whitespace, though since some tools such as lint look at comments, this method is not perfect.

Why Use a Separate Lexer?

Typically, lexical scanning is done in a separate module from the parser, feeding the parser with a stream of input tokens only. Theoretically it is not necessary implement this separation as in the end there is only one set of syntactical rules defining the language, so in theory we could write the whole parser in one module. In fact, Spirit.Qi allows you to write parsers without using a lexer, parsing the input character stream directly, and for the most part this is the way Spirit has been used since its invention.

However, this separation has both practical and theoretical basis, and proves to be very useful in practical applications. In 1956, Noam Chomsky defined the "Chomsky Hierarchy" of grammars:

• Type 0: Unrestricted grammars (e.g., natural languages)

• Type 1: Context-Sensitive grammars

• Type 2: Context-Free grammars

• Type 3: Regular grammars

The complexity of these grammars increases from regular grammars being the simplest to unrestricted grammars being the most complex. Similarly, the complexity of pattern recognition for these grammars increases. Although, a few features of some programming languages (such as C++) are Type 1, fortunately for the most part programming languages can be described using only the Types 2 and 3. The neat part about these two types is that they are well known and the ways to parse them are well understood. It has been shown that any regular grammar can be parsed using a state machine (finite automaton). Similarly, context-free grammars can always be parsed using a push-down automaton (essentially a state machine augmented by a stack).

In real programming languages and practical grammars, the parts that can be handled as regular expressions tend to be the lower-level pieces, such as the definition of an identifier or of an integer value:

```plaintext
letter := [a-zA-Z]
digit := [0-9]
identifier := letter [ letter | digit ]*
in integer := digit+
```

Higher level parts of practical grammars tend to be more complex and can't be implemented using plain regular expressions. We need to store information on the built-in hardware stack while recursing the grammar hierarchy, and that is the preferred approach used for top-down parsing. Since it takes a different kind of abstract machine to parse the two types of grammars, it proved to be efficient to separate the lexical scanner into a separate module which is built around the idea of a state machine. The goal here is to use the simplest parsing technique needed for the job.
Another, more practical, reason for separating the scanner from the parser is the need for backtracking during parsing. The input data is a stream of characters, which is often thought to be processed left to right without any backtracking. Unfortunately, in practice most of the time that isn't possible. Almost every language has certain keywords such as IF, FOR, and WHILE. The decision if a certain character sequence actually comprises a keyword or just an identifier often can be made only after seeing the first delimiter after it. In fact, this makes the process backtracking, since we need to store the string long enough to be able to make the decision. The same is true for more coarse grained language features such as nested IF/ELSE statements, where the decision about to which IF belongs the last ELSE statement can be made only after seeing the whole construct.

So the structure of a conventional compiler often involves splitting up the functions of the lower-level and higher-level parsing. The lexical scanner deals with things at the character level, collecting characters into strings, converting character sequence into different representations as integers, etc., and passing them along to the parser proper as indivisible tokens. It's also considered normal to let the scanner do additional jobs, such as identifying keywords, storing identifiers in tables, etc.

Now, Spirit follows this structure, where Spirit.Lex can be used to implement state machine based recognizers, while Spirit.Qi can be used to build recognizers for context free grammars. Since both modules are seamlessly integrated with each other and with the C++ target language it is even possible to use the provided functionality to build more complex grammar recognizers.

Advantages of using Spirit.Lex

The advantage of using Spirit.Lex to create the lexical analyzer over using more traditional tools such as Flex is its carefully crafted integration with the Spirit library and the C++ host language. You don't need any external tools to generate the code, your lexer will be perfectly integrated with the rest of your program, making it possible to freely access any context information and data structure. Since the C++ compiler sees all the code, it will generate optimal code no matter what configuration options have been chosen by the user. Spirit.Lex gives you the vast majority of features you could get from a similar Flex program without the need to leave C++ as a host language:

- The definition of tokens is done using regular expressions (patterns)
- The token definitions can refer to special substitution strings (pattern macros) simplifying pattern definitions
- The generated lexical scanner may have multiple start states
- It is possible to attach code to any of the token definitions; this code gets executed whenever the corresponding token pattern has been matched

Even if it is possible to use Spirit.Lex to generate C++ code representing the lexical analyzer (we will refer to that as the static model, described in more detail in the section The Static Model) - a model very similar to the way Flex operates - we will mainly focus on the opposite, the dynamic model. You can directly integrate the token definitions into your C++ program, building the lexical analyzer dynamically at runtime. The dynamic model is something not supported by Flex or other lexical scanner generators (such as re2c, Ragel, etc.). This dynamic flexibility allows you to speed up the development of your application.

The Library Structure of Spirit.Lex

The figure below shows a high level overview of how the Spirit.Lex library might be used in an application. Spirit.Lex allows to create lexical analyzers based on patterns. These patterns are regular expression based rules used to define the different tokens to be recognized in the character input sequence. The input sequence is expected to be provided to the lexical analyzer as an arbitrary standard forward iterator. The lexical analyzer itself exposes a standard forward iterator as well. The difference here is that the exposed iterator provides access to the token sequence instead of to the character sequence. The tokens in this sequence are constructed on the fly by analyzing the underlying character sequence and matching this to the patterns as defined by the application.
Spirit.Lex Tutorials

Spirit.Lex Tutorials Overview

The Spirit.Lex library implements several components on top of possibly different lexer generator libraries. It exposes a pair of iterators, which, when dereferenced, return a stream of tokens generated from the underlying character stream. The generated tokens are based on the token definitions supplied by the user.

Currently, Spirit.Lex is built on top of Ben Hanson's excellent Lexertl library (which is a proposed Boost library). Lexertl provides the necessary functionality to build state machines based on a set of supplied regular expressions. But Spirit.Lex is not restricted to be used with Lexertl. We expect it to be usable in conjunction with any other lexical scanner generator library, all what needs to be implemented is a set of wrapper objects exposing a well defined interface as described in this documentation.

Note

For the sake of clarity all examples in this documentation assume Spirit.Lex to be used on top of Lexertl.

Building a lexer using Spirit.Lex is highly configurable, where most of this configuration is done at compile time. Almost all of the configurable parameters have generally useful default values, allowing project startup to be a easy and straightforward task. Here is a (non-complete) list of features you can tweak to adjust the generated lexer instance to the actual needs:

- Select and customize the token type to be generated by the lexer instance.
- Select and customize the token value types the generated token instances will be able to hold.
- Select the iterator type of the underlying input stream, which will be used as the source for the character stream to tokenize.
- Customize the iterator type returned by the lexer to enable debug support, special handling of certain input sequences, etc.
- Select the dynamic or the static runtime model for the lexical analyzer.

Special care has been taken during the development of the library that optimal code will be generated regardless of the configuration options selected.

The series of tutorial examples of this section will guide you through some common use cases helping to understand the big picture. The first two quick start examples (Lex Quickstart 1 - A word counter using Spirit.Lex and Lex Quickstart 2 - A better word counter using Spirit.Lex) introduce the Spirit.Lex library while building two stand alone applications, not being connected to or depending
on any other part of Spirit. The section Lex Quickstart 3 - Counting Words Using a Parser demonstrates how to use a lexer in conjunction with a parser (where obviously the parser is built using Spirit.Qi).

**Quickstart 1 - A word counter using Spirit.Lex**

*Spirit.Lex* is very modular, which follows the general building principle of the *Spirit* libraries. You never pay for features you don't use. It is nicely integrated with the other parts of *Spirit* but nevertheless can be used separately to build stand alone lexical analyzers. The first quick start example describes a stand alone application: counting characters, words, and lines in a file, very similar to what the well known Unix command *wc* is doing (for the full example code see here: word_count Functor.cpp).

**Prerequisites**

The only required #include specific to *Spirit.Lex* follows. It is a wrapper for all necessary definitions to use *Spirit.Lex* in a stand alone fashion, and on top of the *Lexertl* library. Additionally we #include two of the Boost headers to define boost::bind() and boost::ref().

```cpp
#include <boost/spirit/include/lex_lexertl.hpp>
#include <boost/bind.hpp>
#include <boost/ref.hpp>
```

To make all the code below more readable we introduce the following namespaces.

```cpp
namespace lex = boost::spirit::lex;
```

**Defining Tokens**

The most important step while creating a lexer using *Spirit.Lex* is to define the tokens to be recognized in the input sequence. This is normally done by defining the regular expressions describing the matching character sequences, and optionally their corresponding token ids. Additionally the defined tokens need to be associated with an instance of a lexer object as provided by the library. The following code snippet shows how this can be done using *Spirit.Lex*.

```cpp
template <typename Lexer>
struct word_count_tokens : lex::lexer<Lexer>
{
    word_count_tokens()
    {
        // define tokens (the regular expression to match and the corresponding
        // token id) and add them to the lexer
        this->self.add
            ("[^ \t\n]+", ID_WORD) // words (anything except ' ', '	' or '
')
            ("\n", ID_EOL) // newline characters
            (".*", ID_CHAR) // anything else is a plain character
            ;
    }
};
```

**Doing the Useful Work**

We will use a setup, where we want the *Spirit.Lex* library to invoke a given function after any of the generated tokens is recognized. For this reason we need to implement a functor taking at least the generated token as an argument and returning a boolean value all-
lowing to stop the tokenization process. The default token type used in this example carries a token value of the type `boost::iterator_range<BaseIterator>` pointing to the matched range in the underlying input sequence.

In this example the struct 'counter' is used as a functor counting the characters, words and lines in the analyzed input sequence by identifying the matched tokens as passed from the `Spirit.Lex` library.

```cpp
struct counter
{
    // the function operator gets called for each of the matched tokens
    // c, l, w are references to the counters used to keep track of the numbers
    template <typename Token>
    bool operator()(Token const& t, std::size_t& c, std::size_t& w, std::size_t& l) const
    {
        switch (t.id()) {
            case ID_WORD: // matched a word
                // since we're using a default token type in this example, every
                // token instance contains a `iterator_range<BaseIterator>` as its token
                // attribute pointing to the matched character sequence in the input
                ++w; c += t.value().size();
                break;
            case ID_EOL: // matched a newline character
                ++l; ++c;
                break;
            case ID_CHAR: // matched something else
                ++c;
                break;
        }
        return true; // always continue to tokenize
    }
};
```

All what is left is to write some boilerplate code helping to tie together the pieces described so far. To simplify this example we call the `lex::tokenize()` function implemented in `Spirit.Lex` (for a more detailed description of this function see here: FIXME), even if we could have written a loop to iterate over the lexer iterators [first, last) as well.

**Pulling Everything Together**

The main function simply loads the given file into memory (as a `std::string`), instantiates an instance of the token definition template using the correct iterator type (`word_count_tokens<char const*>`), and finally calls `lex::tokenize`, passing an instance of the counter function object. The return value of `lex::tokenize()` will be `true` if the whole input sequence has been successfully tokenized, and `false` otherwise.
```cpp
int main(int argc, char* argv[0])
{
    // these variables are used to count characters, words and lines
    std::size_t c = 0, w = 0, l = 0;

    // read input from the given file
    std::string str (read_from_file(argc == 1 ? "word_count.input" : argv[1]));

    // create the token definition instance needed to invoke the lexical analyzer
    word_count_tokens<lex::lexertl::lexer<> > word_count_functor;

    // tokenize the given string, the bound functor gets invoked for each of
    // the matched tokens
    char const* first = str.c_str();
    char const* last = &first[str.size()];
    bool r = lex::tokenize(first, last, word_count_functor,
                           boost::bind(counter(), _1, boost::ref(c), boost::ref(w), boost::ref(l)));

    // print results
    if (r) {
        std::cout << "lines: " << l << " words: " << w
                  << " characters: " << c << "\n";
    }
    else {
        std::string rest(first, last);
        std::cout << "Lexical analysis failed\nstopped at: \"
                  << rest << "\n";
    }
    return 0;
}

Comparing Spirit.Lex with Flex

This example was deliberately chosen to be as much as possible similar to the equivalent Flex program (see below), which isn’t too different from what has to be written when using Spirit.Lex.

Note

Interestingly enough, performance comparisons of lexical analyzers written using Spirit.Lex with equivalent programs generated by Flex show that both have comparable execution speeds! Generally, thanks to the highly optimized Lexertl library and due its carefully designed integration with Spirit the abstraction penalty to be paid for using Spirit.Lex is negligible.

The remaining examples in this tutorial will use more sophisticated features of Spirit.Lex, mainly to allow further simplification of the code to be written, while maintaining the similarity with corresponding features of Flex. Spirit.Lex has been designed to be as similar to Flex as possible. That is why this documentation will provide the corresponding Flex code for the shown Spirit.Lex examples almost everywhere. So consequently, here is the Flex code corresponding to the example as shown above.
```
Quickstart 2 - A better word counter using SpiritLex

People familiar with Flex will probably complain about the example from the section Lex Quickstart 1 - A word counter using SpiritLex as being overly complex and not being written to leverage the possibilities provided by this tool. In particular the previous example did not directly use the lexer actions to count the lines, words, and characters. So the example provided in this step of the tutorial will show how to use semantic actions in SpiritLex. Even though this examples still counts textual elements, the purpose is to introduce new concepts and configuration options along the lines (for the full example code see here: word_count_lexer.cpp).

Prerequisites

In addition to the only required #include specific to SpiritLex this example needs to include a couple of header files from the Boost.Phoenix library. This example shows how to attach functors to token definitions, which could be done using any type of C++ technique resulting in a callable object. Using Boost.Phoenix for this task simplifies things and avoids adding dependencies to other libraries (Boost.Phoenix is already in use for Spirit anyway).

```cpp
#include <boost/spirit/include/lex_lexertl.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/spirit/include/phoenix_statement.hpp>
#include <boost/spirit/include/phoenix_algorithm.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
```

To make all the code below more readable we introduce the following namespaces.
namespace lex = boost::spirit::lex;

To give a preview at what to expect from this example, here is the flex program which has been used as the starting point. The useful code is directly included inside the actions associated with each of the token definitions.

```c
%{
  int c = 0, w = 0, l = 0;
%
%^ \t\n^+ { ++w; c += yyleng; }
\n    { ++c; ++l; }  
.    { ++c; }    
%
main()
{
  yylex();
  printf("%d %d %d\n", l, w, c);
}
```

Semantic Actions in Spirit.Lex

Spirit.Lex uses a very similar way of associating actions with the token definitions (which should look familiar to anybody knowledgeable with Spirit as well): specifying the operations to execute inside of a pair of [] brackets. In order to be able to attach semantic actions to token definitions for each of them there is defined an instance of a token_def<>.

```cpp
template <typename Lexer>
struct word_count_tokens : lex::lexer<Lexer>
{
  word_count_tokens()
    : c(0), w(0), l(0)
    , word("[^ \t\n]+")   // define tokens
    , eol("\n")
    , any("."
    |
    using boost::spirit::lex::_start;
    using boost::spirit::lex::_end;
    using boost::phoenix::ref;

    // associate tokens with the lexer
    this->self
      = word [++ref(w), ref(c) += distance(_start, _end)]
        | eol [++ref(c), ++ref(l)]
        | any [++ref(c)]
    ;

  std::size_t c, w, l;
  lex::token_def<> word, eol, any;
};
```

The semantics of the shown code is as follows. The code inside the [] brackets will be executed whenever the corresponding token has been matched by the lexical analyzer. This is very similar to Flex, where the action code associated with a token definition gets executed after the recognition of a matching input sequence. The code above uses function objects constructed using Boost.Phoenix, but it is possible to insert any C++ function or function object as long as it exposes the proper interface. For more details on please refer to the section Lexer Semantic Actions.

Associating Token Definitions with the Lexer

If you compare this code to the code from Lex Quickstart 1 - A word counter using Spirit.Lex with regard to the way how token definitions are associated with the lexer, you will notice a different syntax being used here. In the previous example we have been
using the `self.add()` style of the API, while we here directly assign the token definitions to `self`, combining the different token definitions using the `|` operator. Here is the code snippet again:

```cpp
this->self
    = word [++ref(w), ref(c) += distance(_1)]
    | eol [++ref(c), ++ref(l)]
    | any [++ref(c)];
```

This way we have a very powerful and natural way of building the lexical analyzer. If translated into English this may be read as:
The lexical analyzer will recognize (`'='`) tokens as defined by any of (`'|'`) the token definitions `word`, `eol`, and `any`.

A second difference to the previous example is that we do not explicitly specify any token ids to use for the separate tokens. Using semantic actions to trigger some useful work has freed us from the need to define those. To ensure every token gets assigned a id the `Spirit.Lex` library internally assigns unique numbers to the token definitions, starting with the constant defined by `boost::spirit::lex::min_token_id`.

### Pulling everything together

In order to execute the code defined above we still need to instantiate an instance of the lexer type, feed it from some input sequence and create a pair of iterators allowing to iterate over the token sequence as created by the lexer. This code shows how to achieve these steps:

```cpp
int main(int argc, char* argv[]) {

    typedef lex::lexertl::token<char const*, lex::omit, boost::mpl::false_>
        token_type;

    typedef lex::lexertl::actor_lexer<token_type> lexer_type;

    typedef word_count_tokens<lexer_type> word_count_lexer;

    std::string str (read_from_file(1 == argc ? "word_count.input" : argv[1]));
    char const* first = str.c_str();
    char const* last = &first[str.size()];

    lexer_type::iterator_type iter = word_count_lexer.begin(first, last);
    lexer_type::iterator_type end = word_count_lexer.end();

    while (iter != end || token_is_valid(*iter))
        ++iter;

    if (iter == end) {
        std::cout << "lines: " << word_count_lexer.l
                  << ", words: " << word_count_lexer.w
                  << ", characters: " << word_count_lexer.c
                  << "\n";
    } else {
        std::string rest(first, last);
        std::cout << "Lexical analysis failed\n" << "stopped at: \n"
                   << rest << "\n";
    }
    return 0;
}
```

Specifying `omit` as the token attribute type generates a token class not holding any token attribute at all (not even the iterator range of the matched input sequence), therefore optimizing the token, the lexer, and possibly the parser implementation as
much as possible. Specifying mpl::false_ as the 3rd template parameter generates a token type and an iterator, both holding no lexer state, allowing for even more aggressive optimizations. As a result the token instances contain the token ids as the only data member.

1. This defines the lexer type to use
2. Create the lexer object instance needed to invoke the lexical analysis
3. Read input from the given file, tokenize all the input, while discarding all generated tokens
4. Create a pair of iterators returning the sequence of generated tokens
5. Here we simply iterate over all tokens, making sure to break the loop if an invalid token gets returned from the lexer

Quickstart 3 - Counting Words Using a Parser

The whole purpose of integrating Spirit.Lex as part of the Spirit library was to add a library allowing the merger of lexical analysis with the parsing process as defined by a Spirit grammar. Spirit parsers read their input from an input sequence accessed by iterators. So naturally, we chose iterators to be used as the interface between the lexer and the parser. A second goal of the lexer/parser integration was to enable the usage of different lexical analyzer libraries. The utilization of iterators seemed to be the right choice from this standpoint as well, mainly because these can be used as an abstraction layer hiding implementation specifics of the used lexer library. The picture below shows the common flow control implemented while parsing combined with lexical analysis.

Another problem related to the integration of the lexical analyzer with the parser was to find a way how the defined tokens syntactically could be blended with the grammar definition syntax of Spirit. For tokens defined as instances of the token_def<> class the most natural way of integration was to allow to directly use these as parser components. Semantically these parser components succeed matching their input whenever the corresponding token type has been matched by the lexer. This quick start example will demonstrate this (and more) by counting words again, simply by adding up the numbers inside of semantic actions of a parser (for the full example code see here: word_count.cpp).

Prerequisites

This example uses two of the Spirit library components: Spirit.Lex and Spirit.Qi, consequently we have to #include the corresponding header files. Again, we need to include a couple of header files from the Boost.Phoenix library. This example shows how to attach functors to parser components, which could be done using any type of C++ technique resulting in a callable object. Using Boost.Phoenix for this task simplifies things and avoids adding dependencies to other libraries (Boost.Phoenix is already in use for Spirit anyway).

```c++
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/include/lex_lexertl.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/spirit/include/phoenix_statement.hpp>
#include <boost/spirit/include/phoenix_container.hpp>
```

To make all the code below more readable we introduce the following namespaces.
using namespace boost::spirit;
using namespace boost::spirit::ascii;

Defining Tokens

If compared to the two previous quick start examples (Lex Quickstart 1 - A word counter using Spirit.Lex and Lex Quickstart 2 - A better word counter using Spirit.Lex) the token definition class for this example does not reveal any surprises. However, it uses lexer token definition macros to simplify the composition of the regular expressions, which will be described in more detail in the section FIXME. Generally, any token definition is usable without modification from either a stand alone lexical analyzer or in conjunction with a parser.

```cpp
template <typename Lexer>
struct word_count_tokens : lex::lexer<Lexer>
{
    word_count_tokens()
    {
        // define patterns (lexer macros) to be used during token definition
        // below
        this->self.add_pattern
        ("WORD", "[^ \t\n]+")
        ;

        // define tokens and associate them with the lexer
        word = "(WORD)"; // reference the pattern 'WORD' as defined above

        // this lexer will recognize 3 token types: words, newlines, and
        // everything else
        this->self.add
        (word) // no token id is needed here
        ("\n") // characters are usable as tokens as well
        (".*", IDANY) // string literals will not be esacped by the library
        ;

        // the token 'word' exposes the matched string as its parser attribute
        lex::token_def<std::string> word;
    }
};
```

Using Token Definition Instances as Parsers

While the integration of lexer and parser in the control flow is achieved by using special iterators wrapping the lexical analyzer, we still need a means of expressing in the grammar what tokens to match and where. The token definition class above uses three different ways of defining a token:

- Using an instance of a `token_def<>`, which is handy whenever you need to specify a token attribute (for more information about lexer related attributes please look here: Lexer Attributes).

- Using a single character as the token, in this case the character represents itself as a token, where the token id is the ASCII character value.

- Using a regular expression represented as a string, where the token id needs to be specified explicitly to make the token accessible from the grammar level.

All three token definition methods require a different method of grammar integration. But as you can see from the following code snippet, each of these methods are straightforward and blend the corresponding token instances naturally with the surrounding Spirit.Qi grammar syntax.
### Token definition

<table>
<thead>
<tr>
<th>Token definition</th>
<th>Parser integration</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>token_def&lt;&gt;</code></td>
<td>The <code>token_def&lt;&gt;</code> instance is directly usable as a parser component. Parsing of this component will succeed if the regular expression used to define this has been matched successfully.</td>
</tr>
<tr>
<td>Single character</td>
<td>The single character is directly usable in the grammar. However, under certain circumstances it needs to be wrapped by a <code>char_()</code> parser component. Parsing of this component will succeed if the single character has been matched.</td>
</tr>
<tr>
<td>Explicit token id</td>
<td>To use an explicit token id in a Spirit Qi grammar you are required to wrap it with the special <code>token()</code> parser component. Parsing of this component will succeed if the current token has the same token id as specified in the expression <code>token(&lt;id&gt;).</code></td>
</tr>
</tbody>
</table>

The grammar definition below uses each of the three types demonstrating their usage.

```cpp
template <typename Iterator>
struct word_count_grammar : qi::grammar<Iterator>
{
    template <typename TokenDef>
    word_count_grammar(TokenDef const& tok)
        : word_count_grammar::base_type(start), c(0), w(0), l(0)
    {
        using boost::phoenix::ref;
        using boost::phoenix::size;

        start = *( tok.word[++ref(w), ref(c) += size(_1)]
            | lit('
')[++ref(c), ++ref(l)]
            | qi::token(IDANY)[++ref(c)]
        )
    
    std::size_t c, w, l;
    qi::rule<Iterator> start;
};
```

As already described (see: Attributes), the Spirit Qi parser library builds upon a set of of fully attributed parser components. Consequently, all token definitions support this attribute model as well. The most natural way of implementing this was to use the token values as the attributes exposed by the parser component corresponding to the token definition (you can read more about this topic here: About Tokens and Token Values). The example above takes advantage of the full integration of the token values as the `token_def<>`’s parser attributes: the `word` token definition is declared as a `token_def<std::string>`, making every instance of a `word` token carry the string representation of the matched input sequence as its value. The semantic action attached to `tok.word` receives this string (represented by the `_1` placeholder) and uses it to calculate the number of matched characters: `ref(c) += size(_1)`.

### Pulling Everything Together

The main function needs to implement a bit more logic now as we have to initialize and start not only the lexical analysis but the parsing process as well. The three type definitions (typedef statements) simplify the creation of the lexical analyzer and the grammar. After reading the contents of the given file into memory it calls the function `tokenize_and_parse()` to initialize the lexical analysis and parsing processes.
```cpp
int main(int argc, char* argv[]) {
    typedef lex::lexertl::token<
        char const*, boost::mpl::vector<std::string>
    > token_type;
    typedef lex::lexertl::lexer<token_type> lexer_type;
    typedef word_count_tokens<lexer_type>::iterator_type iterator_type;

    // now we use the types defined above to create the lexer and grammar
    // object instances needed to invoke the parsing process
    word_count_tokens<lexer_type> word_count; // Our lexer
    word_count_grammar<iterator_type> g(word_count); // Our parser

    // read in the file int memory
    std::string str(read_from_file(1 == argc ? "word_count.input" : argv[1]));
    char const* first = str.c_str();
    char const* last = &first[str.size()];

    bool r = lex::tokenize_and_parse(first, last, word_count, g);

    if (r) {
        std::cout << "lines: " << g.l << " words: " << g.w
                  << " characters: " << g.c << "\n";
    } else {
        std::cerr << "Parsing failed\n" << "stopped at: \n"
                  << rest << "\n";
    }
    return 0;
}
```

1. Define the token type to be used: `std::string` is available as the type of the token attribute
2. Define the lexer type to use implementing the state machine
3. Define the iterator type exposed by the lexer type
4. Parsing is done based on the the token stream, not the character stream read from the input. The function `tokenize_and_parse()` wraps the passed iterator range `[first, last)` by the lexical analyzer and uses its exposed iterators to parse the toke stream.

## Abstracts

### Lexer Primitives

#### About Tokens and Token Values

As already discussed, lexical scanning is the process of analyzing the stream of input characters and separating it into strings called tokens, most of the time separated by whitespace. The different token types recognized by a lexical analyzer often get assigned unique integer token identifiers (token ids). These token ids are normally used by the parser to identify the current token without having to look at the matched string again. The `Spirit.Lex` library is not different with respect to this, as it uses the token ids as the main means of identification of the different token types defined for a particular lexical analyzer. However, it is different from commonly used lexical analyzers in the sense that it returns (references to) instances of a (user defined) token class to the user. The only limitation of this token class is that it must carry at least the token id of the token it represents. For more information about the interface a user defined token type has to expose please look at the Token Class reference. The library provides a default token type based on the Lexertl library which should be sufficient in most cases: the `lex::lexertl::token<>` type. This section focusses on the description of general features a token class may implement and how this integrates with the other parts of the `Spirit.Lex` library.
The Anatomy of a Token

It is very important to understand the difference between a token definition (represented by the `lex::token_def<>` template) and a token itself (for instance represented by the `lex::lexertl::token<>` template).

The token definition is used to describe the main features of a particular token type, especially:

• to simplify the definition of a token type using a regular expression pattern applied while matching this token type,
• to associate a token type with a particular lexer state,
• to optionally assign a token id to a token type,
• to optionally associate some code to execute whenever an instance of this token type has been matched,
• and to optionally specify the attribute type of the token value.

The token itself is a data structure returned by the lexer iterators. Dereferencing a lexer iterator returns a reference to the last matched token instance. It encapsulates the part of the underlying input sequence matched by the regular expression used during the definition of this token type. Incrementing the lexer iterator invokes the lexical analyzer to match the next token by advancing the underlying input stream. The token data structure contains at least the token id of the matched token type, allowing to identify the matched character sequence. Optionally, the token instance may contain a token value and/or the lexer state this token instance was matched in. The following figure shows the schematic structure of a token.

Figure 8. The structure of a token

The token value and the lexer state the token has been recognized in may be omitted for optimization reasons, thus avoiding the need for the token to carry more data than actually required. This configuration can be achieved by supplying appropriate template parameters for the `lex::lexertl::token<>` template while defining the token type.

The lexer iterator returns the same token type for each of the different matched token definitions. To accommodate for the possible different token value types exposed by the various token types (token definitions), the general type of the token value is a `Boost.Variant`. At a minimum (for the default configuration) this token value variant will be configured to always hold a `boost::iterator_range` containing the pair of iterators pointing to the matched input sequence for this token instance.
If the lexical analyzer is used in conjunction with a *Spirit.Qi* parser, the stored `boost::iterator_range` token value will be converted to the requested token type (parser attribute) exactly once. This happens at the time of the first access to the token value requiring the corresponding type conversion. The converted token value will be stored in the *Boost.Variant* replacing the initially stored iterator range. This avoids having to convert the input sequence to the token value more than once, thus optimizing the integration of the lexer with *Spirit.Qi*, even during parser backtracking.

Here is the template prototype of the `lex::lexertl::token<>` template:

```cpp
template <
    typename Iterator = char const*,
    typename AttributeTypes = mpl::vector0<>,
    typename HasState = mpl::true_
>
struct lexertl_token;
```

where:

- **Iterator**
  - This is the type of the iterator used to access the underlying input stream. It defaults to a plain `char const*`.

- **AttributeTypes**
  - This is either a mpl sequence containing all attribute types used for the token definitions or the type `omit`. If the mpl sequence is empty (which is the default), all token instances will store a `boost::iterator_range<Iterator>` pointing to the start and the end of the matched section in the input stream. If the type is `omit`, the generated tokens will contain no token value (attribute) at all.

- **HasState**
  - This is either `mpl::true_` or `mpl::false_`, allowing control as to whether the generated token instances will contain the lexer state they were generated in. The default is `mpl::true_`, so all token instances will contain the lexer state.

Normally, during construction, a token instance always holds the `boost::iterator_range` as its token value, unless it has been defined using the `omit` token value type. This iterator range then is converted in place to the requested token value type (attribute) when it is requested for the first time.

### The Physiognomy of a Token Definition

The token definitions (represented by the `lex::token_def<>` template) are normally used as part of the definition of the lexical analyzer. At the same time a token definition instance may be used as a parser component in *Spirit.Qi*.

The template prototype of this class is shown here:

```cpp
template<
    typename Attribute = unused_type,
    typename Char = char
>
class token_def;
```

where:

- **Attribute**
  - This is the type of the token value (attribute) supported by token instances representing this token type. This attribute type is exposed to the *Spirit.Qi* library, whenever this token definition is used as a parser component. The default attribute type is `unused_type`, which means the token instance holds a `boost::iterator_range` pointing to the start and the end of the matched section in the input stream. If the attribute is `omit` the token instance will expose no token type at all. Any other type will be used directly as the token value type.

- **Char**
  - This is the value type of the iterator for the underlying input sequence. It defaults to `char`. 
The semantics of the template parameters for the token type and the token definition type are very similar and interdependent. As a rule of thumb you can think of the token definition type as the means of specifying everything related to a single specific token type (such as identifier or integer). On the other hand the token type is used to define the general properties of all token instances generated by the Spirit.Lex library.

**Important**

If you don’t list any token value types in the token type definition declaration (resulting in the usage of the default `boost::iterator_range` token type) everything will compile and work just fine, just a bit less efficient. This is because the token value will be converted from the matched input sequence every time it is requested.

But as soon as you specify at least one token value type while defining the token type you’ll have to list all value types used for `lex::token_def<>` declarations in the token definition class, otherwise compilation errors will occur.

**Examples of using `lex::lexertl::token<>`**

Let’s start with some examples. We refer to one of the Spirit.Lex examples (for the full source code of this example please see example4.cpp).

The first code snippet shows an excerpt of the token definition class, the definition of a couple of token types. Some of the token types do not expose a special token value (`if_`, `else_`, and `while_`). Their token value will always hold the iterator range of the matched input sequence. The token definitions for the `identifier` and the `integer constant` are specialized to expose an explicit token type each: `std::string` and `unsigned int`.

```cpp
// these tokens expose the iterator_range of the matched input sequence
lex::token_def<> if_, else_, while_;

// The following two tokens have an associated attribute type, 'identifier'
// carries a string (the identifier name) and 'constant' carries the
// matched integer value.
//
// Note: any token attribute type explicitly specified in a token_def<>
// declaration needs to be listed during token type definition as
// well (see the typedef for the token_type below).
//
// The conversion of the matched input to an instance of this type occurs
// once (on first access), which makes token attributes as efficient as
// possible. Moreover, token instances are constructed once by the lexer
// library. From this point on tokens are passed by reference only,
// avoiding them being copied around.
lex::token_def<std::string> identifier;
lex::token_def<unsigned int> constant;
```

As the parsers generated by Spirit.Qi are fully attributed, any Spirit.Qi parser component needs to expose a certain type as its parser attribute. Naturally, the `lex::token_def<>` exposes the token value type as its parser attribute, enabling a smooth integration with Spirit.Qi.

The next code snippet demonstrates how the required token value types are specified while defining the token type to use. All of the token value types used for at least one of the token definitions have to be re-iterated for the token definition as well.
// This is the lexer token type to use. The second template parameter lists
// all attribute types used for token_def's during token definition (see
// calculator_tokens<> above). Here we use the predefined lexertl token
// type, but any compatible token type may be used instead.
//
// If you don't list any token attribute types in the following declaration
// (or just use the default token type: lexertl_token<base_iterator_type>)
// it will compile and work just fine, just a bit less efficient. This is
// because the token attribute will be generated from the matched input
// sequence every time it is requested. But as soon as you specify at
// least one token attribute type you'll have to list all attribute types
// used for token_def<> declarations in the token definition class above,
// otherwise compilation errors will occur.
typedef lex::lexertl::token<
    base_iterator_type, boost::mpl::vector<unsigned int, std::string>
> token_type;

To avoid the token to have a token value at all, the special tag omit can be used: token_def<omit> and lex-
ertl_token<base_iterator_type, omit>.

## Tokenizing Input Data

### The tokenize function

The tokenize() function is a helper function simplifying the usage of a lexer in a stand alone fashion. For instance, you may have
a stand alone lexer where all that functional requirements are implemented inside lexer semantic actions. A good example for this
is the word_count_lexer described in more detail in the section Lex Quickstart 2 - A better word counter using Spirit.Lex.

```cpp
template <typename Lexer>
struct word_count_tokens : lex::lexer<Lexer>
{
    word_count_tokens()
        : c(0), w(0), l(0)
        , word("[^ \t\n]+") // define tokens
        , eol("\n")
        , any(".")
    {
        using boost::spirit::lex::_start;
        using boost::spirit::lex::_end;
        using boost::phoenix::ref;

        // associate tokens with the lexer
        this->self
            = word [++ref(w), ref(c) += distance(_start, _end)]
            | eol [++ref(c), ++ref(l)]
            | any [++ref(c)]
    }

    std::size_t c, w, l;
    lex::token_def<> word, eol, any;
};
```

The construct used to tokenize the given input, while discarding all generated tokens is a common application of the lexer. For this
reason Spirit.Lex exposes an API function tokenize() minimizing the code required:
// Read input from the given file
std::string str (read_from_file(1 == argc ? "word_count.input" : argv[1]));

word_count_tokens<lexer_type> word_count_lexer;
std::string::iterator first = str.begin();

// Tokenize all the input, while discarding all generated tokens
bool r = tokenize(first, str.end(), word_count_lexer);

This code is completely equivalent to the more verbose version as shown in the section Lex Quickstart 2 - A better word counter using Spirit.Lex. The function tokenize() will return either if the end of the input has been reached (in this case the return value will be true), or if the lexer couldn't match any of the token definitions in the input (in this case the return value will be false and the iterator first will point to the first not matched character in the input sequence).

The prototype of this function is:

```cpp
template <typename Iterator, typename Lexer>
bool tokenize(Iterator& first, Iterator last, Lexer const& lex,
             typename Lexer::char_type const* initial_state = 0);
```

where:

- **Iterator& first**: The beginning of the input sequence to tokenize. The value of this iterator will be updated by the lexer, pointing to the first not matched character of the input after the function returns.

- **Iterator last**: The end of the input sequence to tokenize.

- **Lexer const& lex**: The lexer instance to use for tokenization.

- **Lexer::char_type const* initial_state**: This optional parameter can be used to specify the initial lexer state for tokenization.

A second overload of the tokenize() function allows specifying of any arbitrary function or function object to be called for each of the generated tokens. For some applications this is very useful, as it might avoid having lexer semantic actions. For an example of how to use this function, please have a look at word_count_functor.cpp:

The main function simply loads the given file into memory (as a std::string), instantiates an instance of the token definition template using the correct iterator type (word_count_tokens<char const*>), and finally calls lex::tokenize, passing an instance of the counter function object. The return value of lex::tokenize() will be true if the whole input sequence has been successfully tokenized, and false otherwise.
```cpp
int main(int argc, char* argv[]) {
// these variables are used to count characters, words and lines
std::size_t c = 0, w = 0, l = 0;

// read input from the given file
std::string str (read_from_file(l == argc ? "word_count.input" : argv[1]));

// create the token definition instance needed to invoke the lexical analyzer
word_count_tokens<lex::lexertl::lexer<> > word_count_functor;

// tokenize the given string, the bound functor gets invoked for each of
// the matched tokens
char const* first = str.c_str();
char const* last = &first[str.size()];
bool r = lex::tokenize(first, last, word_count_functor,
boost::bind(counter(), _1, boost::ref(c), boost::ref(w), boost::ref(l)));

// print results
if (r) {
    std::cout << "lines: " << l << " words: " << w
<< " characters: " << c << "\n";
} else {
    std::string rest(first, last);
    std::cout << "Lexical analysis failed\n" << "stopped at: "
<< rest << "\n";
}
return 0;
}
```

Here is the prototype of this `tokenize()` function overload:

```cpp
template <typename Iterator, typename Lexer, typename F>
bool tokenize(Iterator& first, Iterator last, Lexer const& lex, F f,
             typename Lexer::char_type const* initial_state = 0);
```

where:
- **Iterator& first**: The beginning of the input sequence to tokenize. The value of this iterator will be updated by the lexer, pointing to the first not matched character of the input after the function returns.
- **Iterator last**: The end of the input sequence to tokenize.
- **Lexer const& lex**: The lexer instance to use for tokenization.
- **F f**: A function or function object to be called for each matched token. This function is expected to have the prototype: `bool f(Lexer::token_type);`. The `tokenize()` function will return immediately if `f` returns `false`.
- **Lexer::char_type const* initial_state**: This optional parameter can be used to specify the initial lexer state for tokenization.

### Lexer Semantic Actions

The main task of a lexer normally is to recognize tokens in the input. Traditionally this has been complemented with the possibility to execute arbitrary code whenever a certain token has been detected. `Spirit.Lex` has been designed to support this mode of operation as well. We borrow from the concept of semantic actions for parsers (`Spirit.Qi`) and generators (`Spirit.Karma`). Lexer semantic actions may be attached to any token definition. These are C++ functions or function objects that are called whenever a token definition successfully recognizes a portion of the input. Say you have a token definition `D`, and a C++ function `f`, you can make the lexer call `f` whenever it matches an input by attaching `f`: 

---

398
The expression above links \( f \) to the token definition, \( D \). The required prototype of \( f \) is:

\[
\text{void } f(\text{Iterator} & \text{start}, \text{Iterator} & \text{end}, \text{pass_flag} & \text{matched}, \text{Idtype} & \text{id}, \text{Context} & \text{ctx});
\]

where:

- **Iterator& start**
  
  This is the iterator pointing to the begin of the matched range in the underlying input sequence. The type of the iterator is the same as specified while defining the type of the `lexertl::actor_lexer<...>` (its first template parameter). The semantic action is allowed to change the value of this iterator influencing the matched input sequence.

- **Iterator& end**
  
  This is the iterator pointing to the end of the matched range in the underlying input sequence. The type of the iterator is the same as specified while defining the type of the `lexertl::actor_lexer<...>` (its first template parameter). The semantic action is allowed to change the value of this iterator influencing the matched input sequence.

- **pass_flag& matched**
  
  This value is preinitialized to `pass_normal`. If the semantic action sets it to `pass_fail` this behaves as if the token has not been matched in the first place. If the semantic action sets this to `pass_ignore` the lexer ignores the current token and tries to match a next token from the input.

- **Idtype& id**
  
  This is the token id of the type `Idtype` (most of the time this will be a `std::size_t`) for the matched token. The semantic action is allowed to change the value of this token id, influencing the if of the created token.

- **Context& ctx**
  
  This is a reference to a lexer specific, unspecified type, providing the context for the current lexer state. It can be used to access different internal data items and is needed for lexer state control from inside a semantic action.

When using a C++ function as the semantic action the following prototypes are allowed as well:

\[
\text{void } f(\text{Iterator} & \text{start}, \text{Iterator} & \text{end}, \text{pass_flag} & \text{matched}, \text{Idtype} & \text{id});
\]

\[
\text{void } f(\text{Iterator} & \text{start}, \text{Iterator} & \text{end}, \text{pass_flag} & \text{matched});
\]

\[
\text{void } f(\text{Iterator} & \text{start}, \text{Iterator} & \text{end});
\]

\[
\text{void } f();
\]

**Important**

In order to use lexer semantic actions you need to use type `lexertl::actor_lexer<>` as your lexer class (instead of the type `lexertl::lexer<>` as described in earlier examples).

**The context of a lexer semantic action**

The last parameter passed to any lexer semantic action is a reference to an unspecified type (see the `Context` type in the table above). This type is unspecified because it depends on the token type returned by the lexer. It is implemented in the internals of the iterator type exposed by the lexer. Nevertheless, any context type is expected to expose a couple of functions allowing to influence the behavior of the lexer. The following table gives an overview and a short description of the available functionality.
### Table 8. Functions exposed by any context passed to a lexer semantic action

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator const&amp; get_eoi() const</td>
<td>The function get_eoi() may be used by to access the end iterator of the input stream the lexer has been initialized with.</td>
</tr>
<tr>
<td>void more()</td>
<td>The function more() tells the lexer that the next time it matches a rule, the corresponding token should be appended onto the current token value rather than replacing it.</td>
</tr>
<tr>
<td>Iterator const&amp; less(Iterator const&amp; it, int n)</td>
<td>The function less() returns an iterator positioned to the nth input character beyond the current token start iterator (i.e. by passing the return value to the parameter end it is possible to return all but the first n characters of the current token back to the input stream.</td>
</tr>
<tr>
<td>bool lookahead(std::size_t id)</td>
<td>The function lookahead() can be used to implement lookahead for lexer engines not supporting constructs like flex’ a/b (match a, but only when followed by b). It invokes the lexer on the input following the current token without actually moving forward in the input stream. The function returns whether the lexer was able to match a token with the given token-id id.</td>
</tr>
<tr>
<td>std::size_t get_state() const and void set_state(std::size_t state)</td>
<td>The functions get_state() and set_state() may be used to introspect and change the current lexer state.</td>
</tr>
<tr>
<td>token_value_type get_value() const and void set_value(Value const&amp;)</td>
<td>The functions get_value() and set_value() may be used to introspect and change the current token value.</td>
</tr>
</tbody>
</table>

**Lexer Semantic Actions Using Phoenix**

Even if it is possible to write your own function object implementations (i.e. using Boost.Lambda or Boost.Bind), the preferred way of defining lexer semantic actions is to use Boost.Phoenix. In this case you can access the parameters described above by using the predefined Spirit placeholders:
Table 9. Predefined Phoenix placeholders for lexer semantic actions

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_start</td>
<td>Refers to the iterator pointing to the beginning of the matched input sequence. Any modifications to this iterator value will be reflected in the generated token.</td>
</tr>
<tr>
<td>_end</td>
<td>Refers to the iterator pointing past the end of the matched input sequence. Any modifications to this iterator value will be reflected in the generated token.</td>
</tr>
<tr>
<td>_pass</td>
<td>References the value signaling the outcome of the semantic action. This is pre-initialized to <code>lex::pass_flags::pass_normal</code>. If this is set to <code>lex::pass_flags::pass_fail</code>, the lexer will behave as if no token has been matched, if is set to <code>lex::pass_flags::pass_ignore</code>, the lexer will ignore the current match and proceed trying to match tokens from the input.</td>
</tr>
<tr>
<td>_tokenid</td>
<td>Refers to the token id of the token to be generated. Any modifications to this value will be reflected in the generated token.</td>
</tr>
<tr>
<td>_val</td>
<td>Refers to the value the next token will be initialized from. Any modifications to this value will be reflected in the generated token.</td>
</tr>
<tr>
<td>_state</td>
<td>Refers to the lexer state the input has been match in. Any modifications to this value will be reflected in the lexer itself (the next match will start in the new state). The currently generated token is not affected by changes to this variable.</td>
</tr>
<tr>
<td>_eoi</td>
<td>References the end iterator of the overall lexer input. This value cannot be changed.</td>
</tr>
</tbody>
</table>

The context object passed as the last parameter to any lexer semantic action is not directly accessible while using Boost.Phoenix expressions. We rather provide predefined Phoenix functions allowing to invoke the different support functions as mentioned above. The following table lists the available support functions and describes their functionality:
Table 10. Support functions usable from Phoenix expressions inside lexer semantic actions

<table>
<thead>
<tr>
<th>Plain function</th>
<th>Phoenix function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx.more()</td>
<td>more()</td>
<td>The function more() tells the lexer that the next time it matches a rule, the corresponding token should be appended onto the current token value rather than replacing it.</td>
</tr>
<tr>
<td>ctx.less()</td>
<td>less(n)</td>
<td>The function less() takes a single integer parameter n and returns an iterator positioned to the nth input character beyond the current token start iterator (i.e. by assigning the return value to the placeholder _end it is possible to return all but the first n characters of the current token back to the input stream.</td>
</tr>
<tr>
<td>ctx.lookahead()</td>
<td>lookahead(std::size_t) or lookahead(token_def)</td>
<td>The function lookahead() takes a single parameter specifying the token to match in the input. The function can be used for instance to implement lookahead for lexer engines not supporting constructs like flex’ a/b (match a, but only when followed by b). It invokes the lexer on the input following the current token without actually moving forward in the input stream. The function returns whether the lexer was able to match the specified token.</td>
</tr>
</tbody>
</table>

The Static Lexer Model

The documentation of Spirit.Lex so far mostly was about describing the features of the dynamic model, where the tables needed for lexical analysis are generated from the regular expressions at runtime. The big advantage of the dynamic model is its flexibility, and its integration with the Spirit library and the C++ host language. Its big disadvantage is the need to spend additional runtime to generate the tables, which especially might be a limitation for larger lexical analyzers. The static model strives to build upon the smooth integration with Spirit and C++, and reuses large parts of the Spirit.Lex library as described so far, while overcoming the additional runtime requirements by using pre-generated tables and tokenizer routines. To make the code generation as simple as possible, the static model reuses the token definition types developed for the dynamic model without any changes. As will be shown in this section, building a code generator based on an existing token definition type is a matter of writing 3 lines of code.

Assuming you already built a dynamic lexer for your problem, there are two more steps needed to create a static lexical analyzer using Spirit.Lex:

1. generating the C++ code for the static analyzer (including the tokenization function and corresponding tables), and
2. modifying the dynamic lexical analyzer to use the generated code.

Both steps are described in more detail in the two sections below (for the full source code used in this example see the code here: the common token definition, the code generator, the generated code, and the static lexical analyzer).

But first we provide the code snippets needed to further understand the descriptions. Both, the definition of the used token identifier and the of the token definition class in this example are put into a separate header file to make these available to the code generator and the static lexical analyzer.
The important point here is, that the token definition class is not different from a similar class to be used for a dynamic lexical analyzer. The library has been designed in a way, that all components (dynamic lexical analyzer, code generator, and static lexical analyzer) can reuse the very same token definition syntax.

```cpp
enum tokenids {
    IDANY = boost::spirit::lex::min_token_id + 1,
};
```

The only thing changing between the three different use cases is the template parameter used to instantiate a concrete token definition. For the dynamic model and the code generator you probably will use the `boost::spirit::lexertl::lexer<>` template, where for the static model you will use the `boost::spirit::lexertl::static_lexer<>` type as the template parameter.

This example not only shows how to build a static lexer, but it additionally demonstrates how such a lexer can be used for parsing in conjunction with a `Spirit.Qi` grammar. For completeness, we provide the simple grammar used in this example. As you can see, this grammar does not have any dependencies on the static lexical analyzer, and for this reason it is not different from a grammar used either without a lexer or using a dynamic lexical analyzer as described before.
// This is an ordinary grammar definition following the rules defined by
// Spirit.Qi. There is nothing specific about it, except it gets the token
// definition class instance passed to the constructor to allow accessing the
// embedded token_def<> instances.

template <typename Iterator>
struct word_count_grammar : qi::grammar<Iterator>
{
    template <typename TokenDef>
    word_count_grammar(TokenDef const& tok)
    :
        word_count_grammar::base_type(start), c(0), w(0), l(0)
    {
        using boost::phoenix::ref;
        using boost::phoenix::size;

        // associate the defined tokens with the lexer, at the same time
        // defining the actions to be executed
        start = *( tok.word [ ++ref(w), ref(c) += size(_1) ]
                  | lit('
') [ ++ref(l), ++ref(c) ]
                  | qi::token(IDANY) [ ++ref(c) ]
        );
    }

    std::size_t c, w, l; // counter for characters, words, and lines
    qi::rule<Iterator> start;
};

Generating the Static Analyzer

The first additional step to perform in order to create a static lexical analyzer is to create a small stand alone program for creating
the lexer tables and the corresponding tokenization function. For this purpose the Spirit.Lex library exposes a special API - the
function generate_static_dfa(). It implements the whole code generator, no further code is needed. All what it takes to invoke
this function is to supply a token definition instance, an output stream to use to generate the code to, and an optional string to be
used as a suffix for the name of the generated function. All in all just a couple lines of code.

```cpp
int main(int argc, char* argv[])
{
    // create the lexer object instance needed to invoke the generator
    word_count_tokens<lex::lexertl::lexer<> > word_count; // the token definition

    // open the output file, where the generated tokenizer function will be
    // written to
    std::ofstream out(argc < 2 ? "word_count_static.hpp" : argv[1]);

    // invoke the generator, passing the token definition, the output stream
    // and the name suffix of the tables and functions to be generated
    // The suffix "wc" used below results in a type lexertl::static::lexer_wc
    // to be generated, which needs to be passed as a template parameter to the
    // lexertl::static_lexer template (see word_count_static.cpp).
    lex::lexertl::generate_static_dfa(word_count, out, "wc") ? 0 : -1;
}
```

The shown code generator will generate output, which should be stored in a file for later inclusion into the static lexical analyzer as
shown in the next topic (the full generated code can be viewed here).
Note

The generated code will have compiled in the version number of the current Spirit.Lex library. This version number is used at compilation time of your static lexer object to ensure this is compiled using exactly the same version of the Spirit.Lex library as the lexer tables have been generated with. If the versions do not match you will see an compilation error mentioning an incompatible_static_lexer_version.

Modifying the Dynamic Analyzer

The second required step to convert an existing dynamic lexer into a static one is to change your main program at two places. First, you need to change the type of the used lexer (that is the template parameter used while instantiating your token definition class). While in the dynamic model we have been using the lex::lexertl::lexer<> template, we now need to change that to the lex::lexertl::static_lexer<> type. The second change is tightly related to the first one and involves correcting the corresponding #include statement to:

```
#include <boost/spirit/include/lex_static_lexertl.hpp>
```

Otherwise the main program is not different from an equivalent program using the dynamic model. This feature makes it easy to develop the lexer in dynamic mode and to switch to the static mode after the code has been stabilized. The simple generator application shown above enables the integration of the code generator into any existing build process. The following code snippet provides the overall main function, highlighting the code to be changed.
int main(int argc, char* argv[]) {
    // Define the token type to be used: 'std::string' is available as the type of the token value.
    typedef lex::lexertl::token<
        char const*, boost::mpl::vector<std::string>
    > token_type;

    // Define the lexer type to be used as the base class for our token definition.
    //
    // This is the only place where the code is different from an equivalent dynamic lexical analyzer. We use the `lexertl::static_lexer<>` instead of the `lexertl::lexer<>` as the base class for our token definition type.
    //
    // As we specified the suffix "wc" while generating the static tables we need to pass the type lexertl::static_::lexer_wc as the second template parameter below (see word_count_generate.cpp).
    typedef lex::lexertl::static_lexer<
        token_type, lex::lexertl::static_::lexer_wc
    > lexer_type;

    // Define the iterator type exposed by the lexer.
    typedef word_count_tokens<lexer_type>::iterator_type iterator_type;

    // Now we use the types defined above to create the lexer and grammar object instances needed to invoke the parsing process.
    word_count_tokens<lexer_type> word_count; // Our lexer
    word_count_grammar<iterator_type> g (word_count); // Our parser

    // Read in the file into memory.
    std::string str (read_from_file(1 == argc ? "word_count.input" : argv[1]));
    char const* first = str.c_str();
    char const* last = &first[str.size()];

    // Parsing is done based on the the token stream, not the character stream.
    bool r = lex::tokenize_and_parse(first, last, word_count, g);
    if (r) { // success
        std::cout << "lines: \t" << g.l << ", words: \t" << g.w << ", characters: \t" << g.c << "\n;"
    } else {
        std::string rest(first, last);
        std::cerr << "Parsing failed\n" << "stopped at: \n" << rest << "\n;"
    }
    return 0;
}

Important

The generated code for the static lexer contains the token ids as they have been assigned, either explicitly by the programmer or implicitly during lexer construction. It is your responsibility to make sure that all instances of a particular static lexer type use exactly the same token ids. The constructor of the lexer object has a second (default) parameter allowing it to designate a starting token id to be used while assigning the ids to the token definitions. The requirement above is fulfilled by default as long as no first_id is specified during construction of the static lexer instances.
Quick Reference

This quick reference section is provided for convenience. You can use this section as a sort of a "cheat-sheet" on the most commonly used Lex components. It is not intended to be complete, but should give you an easy way to recall a particular component without having to dig through pages and pages of reference documentation.

Common Notation

Notation

L Lexer type
l, a, b, c, d Lexer objects
Iterator The type of an iterator referring to the underlying input sequence
IdType The token id type
Context The lexer components Context type
ch Character-class specific character (See Character Class Types)
Ch Character-class specific character type (See Character Class Types)
str Character-class specific string (See Character Class Types)
Str Character-class specific string type (See Character Class Types)
Attrib An attribute type
fa A semantic action function with a signature: void f(Iterator&, Iterator&, pass_flag&, Idtype&, Context&).

Primitive Lexer Components

<table>
<thead>
<tr>
<th>Expression</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>n/a</td>
<td>Matches ch</td>
</tr>
<tr>
<td>char_(ch)</td>
<td>n/a</td>
<td>Matches ch</td>
</tr>
<tr>
<td>str</td>
<td>n/a</td>
<td>Matches regular expression str</td>
</tr>
<tr>
<td>string(str)</td>
<td>n/a</td>
<td>Matches regular expression str</td>
</tr>
<tr>
<td>token_def&lt;Attrib&gt;</td>
<td>Attrib</td>
<td>Matches the immediate argument</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>n/a</td>
</tr>
<tr>
<td>l[fa]</td>
<td>Attribute of l Call semantic action fa (after matching l).</td>
<td></td>
</tr>
</tbody>
</table>

Note

The column Attribute in the table above lists the parser attribute exposed by the lexer component if it is used as a parser (see Attribute). A 'n/a' in this columns means the lexer component is not usable as a parser.
Semantics Actions

Has the form:

\[
\text{l}[f]
\]

where \(f\) is a function with the signatures:

- \(\text{void } f();\)
- \(\text{void } f(\text{Iterator}_1, \text{Iterator}_2);\)
- \(\text{void } f(\text{Iterator}_1, \text{Iterator}_2, \text{pass}_\text{flags});\)
- \(\text{void } f(\text{Iterator}_1, \text{Iterator}_2, \text{pass}_\text{flags}, \text{Idtype});\)
- \(\text{void } f(\text{Iterator}_1, \text{Iterator}_2, \text{pass}_\text{flags}, \text{Idtype}, \text{Context});\)

You can use \texttt{Boost.Bind} to bind member functions. For function objects, the allowed signatures are:

- \(\text{void } \texttt{operator}() (\text{unused}_\text{type}, \text{unused}_\text{type}, \text{unused}_\text{type}, \text{unused}_\text{type}, \text{unused}_\text{type}) \text{ const};\)
- \(\text{void } \texttt{operator}() (\text{Iterator}_1, \text{Iterator}_2, \text{unused}_\text{type}, \text{unused}_\text{type}, \text{unused}_\text{type}) \text{ const};\)
- \(\text{void } \texttt{operator}() (\text{Iterator}_1, \text{Iterator}_2, \text{pass}_\text{flags}, \text{unused}_\text{type}, \text{unused}_\text{type}) \text{ const};\)
- \(\text{void } \texttt{operator}() (\text{Iterator}_1, \text{Iterator}_2, \text{pass}_\text{flags}, \text{Idtype}, \text{unused}_\text{type}) \text{ const};\)
- \(\text{void } \texttt{operator}() (\text{Iterator}_1, \text{Iterator}_2, \text{pass}_\text{flags}, \text{Idtype}, \text{Context}) \text{ const};\)

The \texttt{unused}_\text{type} is used in the signatures above to signify 'don't care'.

For more information see \texttt{Lexer Semantic Actions}.

Phoenix

\texttt{Boost.Phoenix} makes it easier to attach semantic actions. You just inline your lambda expressions:

\[
\text{l}[\texttt{phoenix-lambda-expression}]
\]

\texttt{Spirit.Lex} provides some \texttt{Boost.Phoenix} placeholders to access important information from the \texttt{Context} that are otherwise difficult to extract.

\texttt{Spirit.Lex} specific \texttt{Phoenix} placeholders

- \texttt{_start, _end} - Iterators pointing to the begin and the end of the matched input sequence.
- \texttt{_pass} - Assign \texttt{lex::pass}_\texttt{flags}::\texttt{pass}_\texttt{fail} to \_\texttt{pass} to force the current match to fail.
- \texttt{_tokenid} - The token id of the matched token.
- \texttt{_val} - The token value of the matched token.
- \texttt{_state} - The lexer state the token has been matched in.
- \texttt{_eoi} - Iterator referring to the current end of the input sequence.

\textbf{Tip}

All of the placeholders in the list above (except \_\texttt{eoi}) can be changed from the inside of the semantic action allowing to modify the lexer behavior. They are defined in the namespace \texttt{boost::spirit::lex}.

For more information see \texttt{Lexer Semantic Actions}.
Supported Regular Expressions
### Table 11. Regular expressions support

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td>Match any character <code>x</code></td>
</tr>
<tr>
<td><code>.</code></td>
<td>Match any except newline (or optionally <strong>any</strong> character)</td>
</tr>
<tr>
<td><code>...</code></td>
<td>All characters taken as literals between double quotes, except escape sequences</td>
</tr>
<tr>
<td><code>[xyz]</code></td>
<td>A character class; in this case matches <code>x</code>, <code>y</code> or <code>z</code></td>
</tr>
<tr>
<td><code>[abj-oZ]</code></td>
<td>A character class with a range in it; matches <code>a</code>, <code>b</code> any letter from <code>j</code> through <code>o</code> or a <code>Z</code></td>
</tr>
<tr>
<td><code>[^A-Z]</code></td>
<td>A negated character class i.e. any character but those in the class. In this case, any character except an uppercase letter</td>
</tr>
<tr>
<td><code>r*</code></td>
<td>Zero or more <code>r</code>'s (greedy), where <code>r</code> is any regular expression</td>
</tr>
<tr>
<td><code>r*?</code></td>
<td>Zero or more <code>r</code>'s (abstemious), where <code>r</code> is any regular expression</td>
</tr>
<tr>
<td><code>r+</code></td>
<td>One or more <code>r</code>'s (greedy)</td>
</tr>
<tr>
<td><code>r+?</code></td>
<td>One or more <code>r</code>'s (abstemious)</td>
</tr>
<tr>
<td><code>r?</code></td>
<td>Zero or one <code>r</code>'s (greedy), i.e. optional</td>
</tr>
<tr>
<td><code>r??</code></td>
<td>Zero or one <code>r</code>'s (abstemious), i.e. optional</td>
</tr>
<tr>
<td><code>r{2,5}</code></td>
<td>Anywhere between two and five <code>r</code>'s (greedy)</td>
</tr>
<tr>
<td><code>r{2,5}?</code></td>
<td>Anywhere between two and five <code>r</code>'s (abstemious)</td>
</tr>
<tr>
<td><code>r{2,}</code></td>
<td>Two or more <code>r</code>'s (greedy)</td>
</tr>
<tr>
<td><code>r{2,}?</code></td>
<td>Two or more <code>r</code>'s (abstemious)</td>
</tr>
<tr>
<td><code>r{4}</code></td>
<td>Exactly four <code>r</code>'s</td>
</tr>
<tr>
<td><code>{NAME}</code></td>
<td>The macro <code>NAME</code> (see below)</td>
</tr>
<tr>
<td><code>&quot;[xyz]\&quot;foo&quot;</code></td>
<td>The literal string <code>[xyz]&quot;foo</code></td>
</tr>
<tr>
<td><code>\X</code></td>
<td>If <code>X</code> is <code>a</code>, <code>b</code>, <code>e</code>, <code>n</code>, <code>r</code>, <code>f</code>, <code>t</code>, <code>v</code> then the ANSI-C interpretation of <code>\X</code>. Otherwise a literal <code>X</code> (used to escape operators such as <code>*</code>)</td>
</tr>
<tr>
<td><code>\0</code></td>
<td>A NUL character (ASCII code 0)</td>
</tr>
<tr>
<td><code>\123</code></td>
<td>The character with octal value 123</td>
</tr>
<tr>
<td><code>\x2a</code></td>
<td>The character with hexadecimal value <code>2a</code></td>
</tr>
<tr>
<td><code>\cX</code></td>
<td>A named control character <code>X</code></td>
</tr>
<tr>
<td><code>\a</code></td>
<td>A shortcut for Alert (bell).</td>
</tr>
<tr>
<td>Expression</td>
<td>Meaning</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
</tr>
<tr>
<td>\b</td>
<td>A shortcut for Backspace</td>
</tr>
<tr>
<td>\e</td>
<td>A shortcut for ESC (escape character 0x1b)</td>
</tr>
<tr>
<td>\n</td>
<td>A shortcut for newline</td>
</tr>
<tr>
<td>\r</td>
<td>A shortcut for carriage return</td>
</tr>
<tr>
<td>\f</td>
<td>A shortcut for form feed 0x0c</td>
</tr>
<tr>
<td>\t</td>
<td>A shortcut for horizontal tab 0x09</td>
</tr>
<tr>
<td>\v</td>
<td>A shortcut for vertical tab 0x0b</td>
</tr>
<tr>
<td>\d</td>
<td>A shortcut for [0-9]</td>
</tr>
<tr>
<td>\D</td>
<td>A shortcut for [^0-9]</td>
</tr>
<tr>
<td>\s</td>
<td>A shortcut for [\x00\t\n\r\f\v]</td>
</tr>
<tr>
<td>\S</td>
<td>A shortcut for [^\x00\t\n\r\f\v]</td>
</tr>
<tr>
<td>\w</td>
<td>A shortcut for [a-zA-Z0-9_]</td>
</tr>
<tr>
<td>\W</td>
<td>A shortcut for [^a-zA-Z0-9_]</td>
</tr>
</tbody>
</table>

(r) Match an r; parenthesis are used to override precedence (see below)

(?r=s:pattern) apply option 'r' and omit option 's' while interpreting pattern. Options may be zero or more of the characters 'i' or 's'. 'i' means case-insensitive. '-i' means case-sensitive. 's' alters the meaning of the '.' syntax to match any single character whatsoever. '-s' alters the meaning of '.' to match any character except '\n'.

rs The regular expression r followed by the regular expression s (a sequence)

r|s Either an r or and s

^r An r but only at the beginning of a line (i.e. when just starting to scan, or just before a newline has been scanned)

r$ An r but only at the end of a line (i.e. just before a newline)

**Note**

POSIX character classes are not currently supported, due to performance issues when creating them in wide character mode.
Tip

If you want to build tokens for syntaxes that recognize items like quotes (""', ''"), backslash (\), here is example syntax to get you started. The lesson here really is to remember that both c++, as well as regular expressions require escaping with \ for some constructs, which can cascade.

```
quote1 = "'"; // match single ''
quote2 = "\\"; // match single '''
literal_quote1 = "\\"; // match backslash followed by single ''
literal_quote2 = "\\\\"; // match backslash followed by single '''
literal_backslash = "\\\\\\"; // match two backslashes
```

Regular Expression Precedence

- \rs has highest precedence
- \r* has next highest (+, ?, (n,m) have the same precedence as *)
- \r|\s has the lowest precedence

Macros

Regular expressions can be given a name and referred to in rules using the syntax {NAME} where NAME is the name you have given to the macro. A macro name can be at most 30 characters long and must start with a _, or a letter. Subsequent characters can be _, -, a letter or a decimal digit.

Reference

Lexer Concepts

Spirit.Lex components fall into a couple of generalized concepts. The Lexer is the most fundamental concept. All Spirit.Lex components are models of the Lexer concept. PrimitiveLexer, UnaryLexer, and NaryLexer are all refinements of the Lexer concept.

The following sections provide details on these concepts.

Lexer

Description

The Lexer is the most fundamental concept. A Lexer has a member function, collect, that accepts a token definition container Def, and a the name of the lexer state the token definitions of the lexer component need to be added to (a string). It doesn't return anything (return type is void). Each Lexer can represent a specific pattern or algorithm, or it can be a more complex lexer component formed as a composition of other Lexer's. Additionally, a Lexer exposes a member add_actions, that accepts the token definition container Def, while returning nothing (again, the returned type is void).

Notation

\l \( A \) Lexer.

\L [A] Lexer type.

\Def [A] Token definition container type.

\State [A] A type used to represent lexer state names.
Valid Expressions

In the expressions below, the behavior of the lexer component, \( l \), is left unspecified in the base Lexer concept. These are specified in subsequent, more refined concepts and by the actual models thereof.

For any Lexer the following expressions must be valid:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l.\text{collect}(\text{def}, \text{state}, \text{target-state}) )</td>
<td>Add all token definitions provided by this Lexer instance to the lexer state of the token definition container ( \text{def} ). After matching this token, the lexer should be switched into the state ( \text{target-state} ) (optional)</td>
<td>void</td>
</tr>
<tr>
<td>( l.\text{add}_{\text{actions}}(\text{def}) )</td>
<td>Add all semantic actions provided by this Lexer instance to the token definition container ( \text{def} ).</td>
<td>void</td>
</tr>
</tbody>
</table>

Type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{traits::is}_{\text{lexer}}&lt;L&gt;::\text{type}</td>
<td>Metafunction that evaluates to \text{mpl::true} if a certain type, ( L ) is a Lexer, \text{mpl::false} otherwise (See MPL Boolean Constant).</td>
</tr>
</tbody>
</table>

Postcondition

Upon return from \( l.\text{collect} \) the following post conditions should hold:

- On return, \( \text{def} \) holds all token definitions defined in the Lexer, \( l \). This includes all Lexer's contained inside \( l \).

Upon return from \( l.\text{add}_{\text{actions}} \) the following post conditions should hold:

- On return, \( \text{def} \) holds all semantic actions correctly associated with the corresponding token definitions as defined in the Lexer, \( l \). This includes all semantic actions defined by the Lexer's contained inside \( l \).

Models

All lexer components in Spirit.Lex are models of the Lexer concept.

PrimitiveLexer

Description

\text{PrimitiveLexer} is the most basic building block that the client uses to build more complex lexer components.

Refinement of

Lexer
Type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>traits::is_primitive_lexer&lt;L&gt;::type</td>
<td>Metafunction that evaluates to mpl::true_ if a certain type, L is a PrimitiveLexer, mpl::false_ otherwise (See MPL Boolean Constant).</td>
</tr>
</tbody>
</table>

Models

The following lexer components conform to this model:

- character literals (i.e. 'x'), char_<tt>
- string literals (*abc*), std::basic_string<> string

FIXME Add more links to PrimitiveLexer models here.

UnaryLexer

Description

UnaryLexer is a composite lexer component that has a single subject. The UnaryLexer may change the behavior of its subject following the Delegate Design Pattern.

Refinement of

Lexer

Notation

l  A UnaryLexer.
L  A UnaryLexer type.

Valid Expressions

In addition to the requirements defined in Lexer, for any UnaryLexer the following must be met:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>l.subject</td>
<td>Subject lexer component.</td>
<td>Lexer</td>
</tr>
</tbody>
</table>

Type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L::subject_type</td>
<td>The subject lexer component type.</td>
</tr>
<tr>
<td>traits::is_unary_lexer&lt;L&gt;::type</td>
<td>Metafunction that evaluates to mpl::true_ if a certain type, L is a UnaryLexer, mpl::false_ otherwise (See MPL Boolean Constant).</td>
</tr>
</tbody>
</table>

Invariants

For any UnaryLexer, L, the following invariant always holds:

- traits::is Lexer<L::subject_type>::type evaluates to mpl::true_
Models

The following lexer components conform to this model:

- action lexer component (allowing to attach semantic actions)

FIXME Add more links to models of UnaryLexer concept

NaryLexer

Description

NaryLexer is a composite lexer component that has one or more subjects. The NaryLexer allows its subjects to be treated in the same way as a single instance of a Lexer following the Composite Design Pattern.

Refinement of

Lexer

Notation

- \(L\) A NaryLexer.
- \(L\) A NaryLexer type.

Valid Expressions

In addition to the requirements defined in Lexer, for any NaryLexer the following must be met:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(l).elements</td>
<td>The tuple of elements.</td>
<td>A Boost.Fusion Sequence of Lexer types.</td>
</tr>
</tbody>
</table>

Type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(l).elements_type</td>
<td>Elements tuple type.</td>
</tr>
<tr>
<td>traits::is_nary Lexer&lt;(L)&gt;::type</td>
<td>Metafunction that evaluates to mpl::true_ if a certain type, (L) is a NaryLexer, mpl::false_ otherwise (See MPL Boolean Constant).</td>
</tr>
</tbody>
</table>

Invariants

For each element, \(E\), in any NaryLexer, \(L\), the following invariant always holds:

- traits::is Lexer<E>::type evaluates to mpl::true_

Models

The following lexer components conform to this model:

- lexer sequence component

FIXME Add more links to models of NaryLexer concept
**Basics**

**Examples**

All sections in the reference present some real world examples. The examples use a common test harness to keep the example code as minimal and direct to the point as possible. The test harness is presented below.

Some includes:

```cpp
#include <boost/spirit/include/lex.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <iostream>
#include <string>
```

Our test functions:

This one tests token definitions.

**Models**

Predefined models include:

- any literal string, e.g. "Hello, World"
- a pointer/reference to a null-terminated array of characters
- `std::basic_string<Char>`

The namespace `boost::spirit::traits` is open for users to provide their own specializations.

**Lexer API**

**Description**

The library provides a couple of free functions to make using the lexer a snap. These functions have three forms. The first form, `tokenize`, simplifies the usage of a stand alone lexer (without parsing). The second form, `tokenize_and_parse`, combines a lexer step with parsing on the token level (without a skipper). The third, `tokenize_and_phrase_parse`, works on the token level as well, but additionally employs a skip parser. The latter two versions can take in attributes by reference that will hold the parsed values on a successful parse.

**Header**

```cpp
// forwards to <boost/spirit/home/lex/tokenize_and_parse.hpp>
#include <boost/spirit/include/lex_tokenize_and_parse.hpp>
```

For variadic attributes:

```cpp
// forwards to <boost/spirit/home/lex/tokenize_and_parse_attr.hpp>
#include <boost/spirit/include/lex_tokenize_and_parse_attr.hpp>
```

The variadic attributes version of the API allows one or more attributes to be passed into the API functions. The functions taking two or more attributes are usable when the parser expression is a `Sequence` only. In this case each of the attributes passed have to match the corresponding part of the sequence.

Also, see Include Structure.
## namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::lex::tokenize</td>
</tr>
<tr>
<td>boost::spirit::lex::tokenize_and_parse</td>
</tr>
<tr>
<td>boost::spirit::lex::tokenize_and_phrase_parse</td>
</tr>
<tr>
<td>boost::spirit::qi::skip_flag::postskip</td>
</tr>
<tr>
<td>boost::spirit::qi::skip_flag::dont_postskip</td>
</tr>
</tbody>
</table>

## synopsis

The `tokenize` function is one of the main lexer API functions. It simplifies using a lexer to tokenize a given input sequence. Its main purpose is to use the lexer to tokenize all the input.

Both functions take a pair of iterators spanning the underlying input stream to scan, the lexer object (built from the token definitions), and an (optional) functor being called for each of the generated tokens. If no function object `f` is given, the generated tokens will be discarded.

The functions return `true` if the scanning of the input succeeded (the given input sequence has been successfully matched by the given token definitions).

The argument `f` is expected to be a function (callable) object taking a single argument of the token type and returning a bool, indicating whether the tokenization should be canceled. If it returns `false` the function `tokenize` will return `false` as well.

The `initial_state` argument forces lexing to start with the given lexer state. If this is omitted lexing starts in the "INITIAL" state.

```cpp
template <typename Iterator, typename Lexer>
inline bool tokenize(
    Iterator& first,
    Iterator last,
    Lexer const& lex,
    typename Lexer::char_type const* initial_state = 0);

template <typename Iterator, typename Lexer, typename F>
inline bool tokenize(
    Iterator& first,
    Iterator last,
    Lexer const& lex,
    F f,
    typename Lexer::char_type const* initial_state = 0);
```

The `tokenize_and_parse` function is one of the main lexer API functions. It simplifies using a lexer as the underlying token source while parsing a given input sequence.

The functions take a pair of iterators spanning the underlying input stream to parse, the lexer object (built from the token definitions) and a parser object (built from the parser grammar definition). Additionally they may take the attributes for the parser step.

The function returns `true` if the parsing succeeded (the given input sequence has been successfully matched by the given grammar).
The functions `tokenize_and_phrase_parse` take a pair of iterators spanning the underlying input stream to parse, the lexer object (built from the token definitions) and a parser object (built from the parser grammar definition). The additional skipper parameter will be used as the skip parser during the parsing process. Additionally they may take the attributes for the parser step.

The function returns `true` if the parsing succeeded (the given input sequence has been successfully matched by the given grammar).

The maximum number of supported arguments is limited by the preprocessor constant `SPIRIT_ARGUMENTS_LIMIT`. This constant defaults to the value defined by the preprocessor constant `PHOENIX_LIMIT` (which in turn defaults to 10).
Note
The variadic function with two or more attributes internally combine references to all passed attributes into a fusion::vector and forward this as a combined attribute to the corresponding one attribute function.

The tokenize_and_phrase_parse functions not taking an explicit skip_flag as one of their arguments invoke the passed skipper after a successful match of the parser expression. This can be inhibited by using the other versions of that function while passing skip_flag::dont_postskip to the corresponding argument.

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForwardIterator</td>
<td>Iterator pointing to the underlying input sequence to parse.</td>
</tr>
<tr>
<td>Lexer</td>
<td>A lexer (token definition) object.</td>
</tr>
<tr>
<td>F</td>
<td>A function object called for each generated token.</td>
</tr>
<tr>
<td>ParserExpr</td>
<td>An expression that can be converted to a Qi parser.</td>
</tr>
<tr>
<td>Skipper</td>
<td>Parser used to skip white spaces.</td>
</tr>
<tr>
<td>Attr1, Attr2, ... , AttrN</td>
<td>One or more attributes.</td>
</tr>
</tbody>
</table>

Token definition Primitives

This module includes different primitives allowing you to create token definitions. It includes char_, character literals, string, and string literals.

Module Headers

```cpp
// forwards to <boost/spirit/home/lex/primitives.hpp>
#include <boost/spirit/include/lex_primitives.hpp>
```

Also, see Include Structure.

Tokens Matching Single Characters

Description

The character based token definitions described in this section are:

The char_ creates token definitions matching single characters. The char_ token definition is associated standard encoding namespace. This is needed when doing basic operations such as forcing lower or upper case and dealing with character ranges.

Header

Module Headers

```cpp
// forwards to <boost/spirit/home/lex/lexer/char_token_def.hpp>
#include <boost/spirit/include/lex_char_token_def.hpp>
```

Also, see Include Structure.
Namespace

Name

lex::char_

Model of

PrimitiveLexer

Notation

ch Character-class specific character from standard character set.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in PrimitiveLexer.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>Create a token definition matching the character literal ch.</td>
</tr>
<tr>
<td>lex::char_(ch)</td>
<td>Create a token definition matching the character ch.</td>
</tr>
</tbody>
</table>

Example
Advanced

In Depth

Parsers in Depth

This section is not for the faint of heart. In here, are distilled the inner workings of Spirit.Qi parsers, using real code from the Spirit library as examples. On the other hand, here is no reason to fear reading on, though. We tried to explain things step by step while highlighting the important insights.

The Parser class is the base class for all parsers.

```cpp
template <typename Derived>
struct parser
{
  struct parser_id;
  typedef Derived derived_type;
  typedef qi::domain domain;

  // Requirement: p.parse(f, l, context, skip, attr) -> bool
  //            p: a parser
  //            f, l: first/last iterator pair
  //            context: enclosing rule context (can be unused_type)
  //            skip: skipper (can be unused_type)
  //            attr: attribute (can be unused_type)

  // Requirement: p.what(context) -> info
  //             p: a parser
  //             context: enclosing rule context (can be unused_type)

  // Requirement: P::template attribute<Ctx, Iter>::type
  //         P: a parser type
  //        Ctx: A context type (can be unused_type)
  //      Iter: An iterator type (can be unused_type)

  Derived const& derived() const
  {
    return *static_cast<Derived const*>(this);
  }
};
```

The Parser class does not really know how to parse anything but instead relies on the template parameter Derived to do the actual parsing. This technique is known as the "Curiously Recurring Template Pattern" in template meta-programming circles. This inheritance strategy gives us the power of polymorphism without the virtual function overhead. In essence this is a way to implement compile time polymorphism.

The Derived parsers, PrimitiveParser, UnaryParser, BinaryParser and NaryParser provide the necessary facilities for parser detection, introspection, transformation and visitation.

Derived parsers must support the following:

```cpp
bool parse(f, l, context, skip, attr)
```

f, l first/last iterator pair
context enclosing rule context (can be unused_type)
The `parse` is the main parser entry point. `skipper` can be an `unused_type`. It's a type used everywhere in Spirit to signify "don't-care". There is an overload for `skip` for `unused_type` that is simply a no-op. That way, we do not have to write multiple parse functions for phrase and character level parsing.

Here are the basic rules for parsing:

- The parser returns `true` if successful, `false` otherwise.
- If successful, `first` is incremented $N$ number of times, where $N$ is the number of characters parsed. $N$ can be zero -- an empty (epsilon) match.
- If successful, the parsed attribute is assigned to `attr`
- If unsuccessful, `first` is reset to its position before entering the parser function. `attr` is untouched.

```cpp
def what(context)
```

The `what` function should be obvious. It provides some information about "what" the parser is. It is used as a debugging aid, for example.

```cpp
P::template attribute<context>::type
```

For our dissection study, we will use a Spirit primitive, the `any_int_parser` in the boost::spirit::qi namespace.

```cpp
[primitive_parsers_any_int_parser]
```

The `any_int_parser` is derived from a `PrimitiveParser<Derived>`, which in turn derives from `parser<Derived>`. Therefore, it supports the following requirements:

- The `parse` member function
- The `what` member function
- The nested `attribute` metafunction

`parse` is the main entry point. For primitive parsers, our first thing to do is call:
to do a pre-skip. After pre-skipping, the parser proceeds to do its thing. The actual parsing code is placed in `extract_int<T, Radix, MinDigits, MaxDigits>::call(first, last, attr);`

This simple no-frills protocol is one of the reasons why `Spirit` is fast. If you know the internals of `Spirit.Classic` and perhaps even wrote some parsers with it, this simple `Spirit` mechanism is a joy to work with. There are no scanners and all that crap.

The `what` function just tells us that it is an integer parser. Simple.

The `attribute` metafunction returns the T template parameter. We associate the `any_int_parser` to some placeholders for `short_`, `int_`, `long_` and `long_long` types. But, first, we enable these placeholders in namespace `boost::spirit`:

```cpp
template <> // enables short_
struct use_terminal<qi::domain, tag::short_> : mpl::true_ {};

template <> // enables int_
struct use_terminal<qi::domain, tag::int_> : mpl::true_ {};

template <> // enables long_
struct use_terminal<qi::domain, tag::long_> : mpl::true_ {};

template <> // enables long_long
struct use_terminal<qi::domain, tag::long_long> : mpl::true_ {};
```

Notice that `any_int_parser` is placed in the namespace `boost::spirit::qi` while these `enablers` are in namespace `boost::spirit`. The reason is that these placeholders are shared by other `Spirit` domains. `Spirit.Qi`, the parser is one domain. `Spirit.Karma`, the generator is another domain. Other parser technologies may be developed and placed in yet another domain. Yet, all these can potentially share the same placeholders for interoperability. The interpretation of these placeholders is domain-specific.

Now that we enabled the placeholders, we have to write generators for them. The `make_xxx` stuff (in `boost::spirit::qi` namespace):

```cpp
template <
    typename T,
    unsigned Radix = 10,
    unsigned MinDigits = 1,
    int MaxDigits = -1>
struct make_int
{
    typedef any_int_parser<T, Radix, MinDigits, MaxDigits> result_type;
    result_type operator() (unused_type, unused_type) const
    {
        return result_type();
    }
};
```

This one above is our main generator. It's a simple function object with 2 (unused) arguments. These arguments are

1. The actual terminal value obtained by proto. In this case, either a `short_`, `int_`, `long_` or `long_long`. We don't care about this.
2. Modifiers. We also don't care about this. This allows directives such as `no_case[p]` to pass information to inner parser nodes. We'll see how that works later.

Now:
These, specialize `qi::make_primitive` for specific tags. They all inherit from `make_int` which does the actual work.

**Composite Parsers**

Let me present the kleene star (also in namespace `spirit::qi`):
template <typename Subject>
struct kleene : unary_parser<kleene<Subject> >
{
  typedef Subject subject_type;

template <typename Context, typename Iterator>
struct attribute
{
  // Build a std::vector from the subject's attribute. Note
  // that build_std_vector may return unused_type if the
  // subject's attribute is an unused_type.
  typedef typename
    traits::build_std_vector<
      typename traits::
        attribute_of<Subject, Context, Iterator>::type
    >::type
type;
};

kleene(Subject const& subject)
  : subject(subject) {}

template <typename F>
bool parse_container(F f) const
{
  while (!f(subject))
    return true;
}

template <typename Iterator, typename Context
  , typename Skipper, typename Attribute>
bool parse(Iterator& first, Iterator const& last
  , Context& context, Skipper const& skipper
  , Attribute& attr) const
{
  // ensure the attribute is actually a container type
  traits::make_container(attr);

  typedef detail::fail_function<Iterator, Context, Skipper>
  fail_function;

  Iterator iter = first;
  fail_function f(iter, last, context, skipper);
  parse_container(detail::make_pass_container(f, attr));

  first = f.first;
  return true;
}

template <typename Context>
info what(Context& context) const
{
  return info("kleene", subject.what(context));
}

Subject subject;
};

Looks similar in form to its primitive cousin, the int_parser. And, again, it has the same basic ingredients required by Derived.

- The nested attribute metafunction
The parse member function

The what member function

kleene is a composite parser. It is a parser that composes another parser, its “subject”. It is a UnaryParser and subclasses from it. Like PrimitiveParser, UnaryParser<Derived> derives from parser<Derived>.

UnaryParser<Derived>, has these expression requirements on Derived:

• p.subject -> subject parser (p is a UnaryParser parser.)
• P::subject_type -> subject parser type (P is a UnaryParser type.)

_parse is the main parser entry point. Since this is not a primitive parser, we do not need to call qi::skip(first, last, skipper).

The subject, if it is a primitive, will do the pre-skip. If it is another composite parser, it will eventually call a primitive parser somewhere down the line which will do the pre-skip. This makes it a lot more efficient than Spirit.Classic. Spirit.Classic puts the skipping business into the so-called ”scanner” which blindly attempts a pre-skip every time we increment the iterator.

What is the attribute of the kleene? In general, it is a std::vector<T> where T is the attribute of the subject. There is a special case though. If T is an unused_type, then the attribute of kleene is also unused_type. traits::build_std_vector takes care of that minor detail.

So, let's parse. First, we need to provide a local attribute of for the subject:

```cpp
typename traits::attribute_of<Subject, Context>::type val;
```

traits::attribute_of<Subject, Context> simply calls the subject's struct attribute<Context> nested metafunction.

val starts out default initialized. This val is the one we'll pass to the subject's parse function.

The kleene repeats indefinitely while the subject parser is successful. On each successful parse, we push_back the parsed attribute to the kleene's attribute, which is expected to be, at the very least, compatible with a std::vector. In other words, although we say that we want our attribute to be a std::vector, we try to be more lenient than that. The caller of kleene's parse may pass a different attribute type. For as long as it is also a conforming STL container with push_back, we are ok. Here is the kleene loop:

```cpp
while (subject.parse(first, last, context, skipper, val))
{
    // push the parsed value into our attribute
    traits::push_back(attr, val);
    traits::clear(val);
}
return true;
```

Take note that we didn't call attr.push_back(val). Instead, we called a Spirit provided function:

```cpp
traits::push_back(attr, val);
```

This is a recurring pattern. The reason why we do it this way is because attr can be unused_type. traits::push_back takes care of that detail. The overload for unused_type is a no-op. Now, you can imagine why Spirit is fast! The parsers are so simple and the generated code is as efficient as a hand rolled loop. All these parser compositions and recursive parse invocations are extensively inlined by a modern C++ compiler. In the end, you get a tight loop when you use the kleene. No more excess baggage. If the attribute is unused, then there is no code generated for that. That's how Spirit is designed.

The what function simply wraps the output of the subject in a "kleene"...

Ok, now, like the int_parser, we have to hook our parser to the qi engine. Here's how we do it:

First, we enable the prefix star operator. In proto, it's called the "dereference":
This is done in namespace boost::spirit like its friend, the use_terminal specialization for our int_parser. Obviously, we use use_operator to enable the dereference for the qi::domain.

Then, we need to write our generator (in namespace qi):

```cpp
template <typename Elements, typename Modifiers>
struct make_composite<proto::tag::dereference, Elements, Modifiers>
  : make_unary_composite<Elements, kleene>
{);
```

This essentially says; for all expressions of the form: *p, to build a kleene parser. Elements is a Boost.Fusion sequence. For the kleene, which is a unary operator, expect only one element in the sequence. That element is the subject of the kleene.

We still don't care about the Modifiers. We'll see how the modifiers is all about when we get to deep directives.

## Customization of Spirit's Attribute Handling

### Why do we need Attribute Customization Points

**Important**

Before you read on please be aware that the interfaces described in this section are not finalized and may change in the future without attempting to be backwards compatible. We document the customization point interfaces anyways as we think they are important. Understanding customization points helps understanding Spirit. Additionally they prove to be powerful tools enabling full integration of the user's data structures with Qi's parsers and Karma's generators.

**Spirit** has been written with extensibility in mind. It provides many different attribute customization points allowing to integrate custom data types with the process of parsing in Spirit.Qi or output generation with Spirit.Karma. All attribute customization points are exposed using a similar technique: full or partial template specialization. **Spirit** generally implements the main template, providing a default implementation. You as the user have to provide a partial or full specialization of this template for the data types you want to integrate with the library. In fact, the library uses these customization points itself for instance to handle the magic of the unused_type attribute type.

Here is an example showing the traits::container_value customization point used by different parsers (such as Kleene, Plus, etc.) to find the attribute type to be stored in a supplied STL container:

```cpp
template <typename Container, typename Enable/* = void*/>
struct container_value
  : detail::remove_value_const<typename Container::value_type>
{);
```

This template is instantiated by the library at the appropriate places while using the supplied container type as the template argument. The embedded type is used as the attribute type while parsing the elements to be store in that container.

The following example shows the predefined specialization for unused_type:
which defines its embedded type to be unused_type as well, this way propagating the 'don't care' attribute status to the embedded parser.

All attribute customization points follow the same scheme. The last template parameter is always typename Enable = void allowing to apply SFINAE for fine grained control over the template specialization process. But most of the time you can safely forget about its existence.

The following sections will describe all customization points, together with a description which needs to be specialized for what purpose.

The Usage of Customization Points

The different customizations points are used by different parts of the library. Part of the customizations points are used by both, Spirit.Qi and Spirit.Karma, whereas others are specialized to be applied for one of the sub-libraries only. We will explain when a specific customization point needs to be implemented and, equally important, which customization points need to be implemented at the same time. Often it is not sufficient to provide a specialization for one single customization point only, in this case you as the user have to provide all necessary customizations for your data type you want to integrate with the library.

Determine if a Type Should be Treated as a Container (Qi and Karma)

is_container

The template is_container is a template meta-function used as an attribute customization point. It is invoked by the Qi Sequence (>>) and Karma Sequence (<<) operators in order to determine whether a supplied attribute can potentially be treated as a container.

Header

```
#include <boost/spirit/home/support/container.hpp>
```

Also, see Include Structure.

Note

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.
**Synopsis**

```cpp
template <typename Container, typename Enable>
struct is_container
{
    <unspecified>;
};
```

**Template parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td>The type <code>Container</code> which needs to be tested whether it has to be treated as a container</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of <code>is_container</code> utilizing SFINAE (i.e. <code>boost::enable_if</code> or <code>boost::disable_if</code>).</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notation**

- `C` A type to be tested whether it needs to be treated as a container.
- `T1, T2, ...` Arbitrary types

**Expression Semantics**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_container&lt;C&gt;::type</code></td>
<td>Result of the metafunction that evaluates to <code>mpl::true_</code> if a given type, <code>C</code>, is to be treated as a container, <code>mpl::false_</code> otherwise. Generally, any implementation of <code>is_container</code> needs to behave as if if was a MPL Boolean Constant.</td>
</tr>
</tbody>
</table>

**Predefined Specializations**

`Spirit` predefines specializations of this customization point for several types. The following table lists those types together with the conditions for which the corresponding specializations will evaluate to `mpl::true_` (see MPL Boolean Constant):

<table>
<thead>
<tr>
<th>Template Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T</code></td>
<td>Returns <code>mpl::true_</code> if <code>T</code> has the following embedded types defined: <code>value_type</code>, <code>iterator</code>, <code>size_type</code>, and <code>reference</code>. Otherwise it will return <code>mpl::false_</code>.</td>
</tr>
<tr>
<td><code>boost::optional&lt;T&gt;</code></td>
<td>Returns <code>is_container&lt;T&gt;::type</code></td>
</tr>
<tr>
<td><code>boost::variant&lt;T1, T2, ...&gt;</code></td>
<td>Returns <code>mpl::true_</code> if at least one of the <code>is_container&lt;TN&gt;::type</code> returns <code>mpl::true_</code> (where <code>TN</code> is <code>T1, T2, ...</code>). Otherwise it will return <code>mpl::false_</code>.</td>
</tr>
<tr>
<td><code>unused_type</code></td>
<td>Returns <code>mpl::false_</code>.</td>
</tr>
</tbody>
</table>
**When to implement**

The customization point `is_container` needs to be implemented for a specific type whenever this type is to be used as an attribute in place of a STL container. It is applicable for parsers (`Spirit.Qi`) and generators (`Spirit.Karma`). As a rule of thumb: it has to be implemented whenever a certain type is to be passed as an attribute to a parser or a generator normally exposing a STL container, and if the type does not expose the interface of a STL container (i.e. `is_container<C>::type` would normally return `mpl::false_`). These components have an attribute propagation rule in the form:

\[
\text{a: A} \rightarrow \text{Op(a): vector<A>}
\]

where `Op(a)` stands for any meaningful operation on the component `a`.

**Related Attribute Customization Points**

If this customization point is implemented, the following other customization points might need to be implemented as well.

<table>
<thead>
<tr>
<th>Name</th>
<th>When to implement</th>
</tr>
</thead>
<tbody>
<tr>
<td>traits::container_value</td>
<td>Needs to be implemented whenever <code>is_container</code> is implemented.</td>
</tr>
<tr>
<td>traits::push_back_container</td>
<td>Qi: List, Kleene, Plus, Repeat.</td>
</tr>
<tr>
<td>traits::container_iterator</td>
<td>Karma: List(%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td>traits::begin_container</td>
<td>Karma: List(%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td>traits::end_container</td>
<td>Karma: List(%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td>traits::dereff_iterator</td>
<td>Karma: List(%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td>traits::next_iterator</td>
<td>Karma: List(%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td>traits::compare_iterators</td>
<td>Karma: List(%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td>traits::clear_value</td>
<td>Qi: List, Kleene, Plus, Repeat.</td>
</tr>
</tbody>
</table>

**Example**

For examples of how to use the customization point `is_container` please see here: embedded_container_example, use_as_container, and counter_example.

**Determine if a Type Should be Treated as a String (Qi and Karma)**

**is_string**

The `is_string` customization point is a template meta-function. It is used by `Qi String Literals (lit(str))`, `Qi Character Literals (lit(c))`, `Karma String Literals (lit(str))`, `Karma Character Literals (lit(c))` and other Spirit components. It determines whether a supplied type can be treated as a string.

**Module Headers**

```cpp
#include <boost/spirit/home/support/string_traits.hpp>
```

Also, see Include Structure.
Note
This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

Namespace

Name

boost::spirit::traits

Synopsis

```cpp
template <typename T>
struct is_string
{
    <unspecified>;
};
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>The type, T which needs to be tested as a string</td>
<td>none</td>
</tr>
</tbody>
</table>

Notation

T  An arbitrary type.
N  An arbitrary integral constant.
Char  A character type.
Traits  A character traits type.
Allocator  A standard allocator type.

Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_string&lt;T&gt;::type</td>
<td>Result of the metafunction that evaluates to mpl::true_ if a given type, T, is to be treated as a string and mpl::false_ otherwise. Generally, any implementation of is_string needs to behave as if it was a MPL Boolean Constant.</td>
</tr>
</tbody>
</table>
**Predefined Specializations**

<table>
<thead>
<tr>
<th>Type</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Returns mpl::false_</td>
</tr>
<tr>
<td>T const</td>
<td>Returns is_string&lt;T&gt;</td>
</tr>
<tr>
<td>char const*</td>
<td>Returns mpl::true_</td>
</tr>
<tr>
<td>wchar_t const*</td>
<td>Returns mpl::true_</td>
</tr>
<tr>
<td>char*</td>
<td>Returns mpl::true_</td>
</tr>
<tr>
<td>wchar_t*</td>
<td>Returns mpl::true_</td>
</tr>
<tr>
<td>char[N]</td>
<td>Returns mpl::true_</td>
</tr>
<tr>
<td>wchar_t[N]</td>
<td>Returns mpl::true_</td>
</tr>
<tr>
<td>char const[N]</td>
<td>Returns mpl::true_</td>
</tr>
<tr>
<td>wchar_t const[N]</td>
<td>Returns mpl::true_</td>
</tr>
<tr>
<td>char(&amp;)[N]</td>
<td>Returns mpl::true_</td>
</tr>
<tr>
<td>wchar_t(&amp;)[N]</td>
<td>Returns mpl::true_</td>
</tr>
<tr>
<td>char const(&amp;)[N]</td>
<td>Returns mpl::true_</td>
</tr>
<tr>
<td>wchar_t const(&amp;)[N]</td>
<td>Returns mpl::true_</td>
</tr>
<tr>
<td>std::basic_string&lt;Char, Traits, Allocator&gt;</td>
<td>Returns mpl::true_</td>
</tr>
</tbody>
</table>

**When to implement**

This customization point needs to be implemented to use user-defined string classes that do not correspond to std::string syntax and semantics.

**Related Attribute Customization Points**

If this customization point is implemented, the following other customization points need to be implemented as well.

<table>
<thead>
<tr>
<th>Name</th>
<th>When to implement</th>
</tr>
</thead>
<tbody>
<tr>
<td>traits::is_char</td>
<td>For string types whose underlying character type is not char or wchar_t, is_char must be implemented.</td>
</tr>
<tr>
<td>traits::char_type_of</td>
<td>Whenever is_string is implemented.</td>
</tr>
<tr>
<td>traits::extract_c_string</td>
<td>Whenever is_string is implemented.</td>
</tr>
</tbody>
</table>
Determine Whether a Component Handles Container Attributes (Qi and Karma)

handles_container

The template handles_container is a template meta-function used as an attribute customization point. It is invoked by the Qi Sequence (>>) and Karma Sequence (<<) operators in order to determine whether a sequence element (component) handles container attributes directly. This customization point is invoked for container attributes only, and only if the sequence is compatible with the supplied container attribute.

If a component, which is part of a sequence is able to handle a container attribute directly, the sequence passes the attribute to the component without any additional action. In Spirit.Qi the component uses the attribute to directly store all matched attributes. In Spirit.Karma the generator component extracts the attributes needed for output generation directly from this attribute.

If a component, which is part of a sequence is not able to handle container attributes, in Spirit.Qi the sequence passes a new instance of the container attributes’ value_type to the parser component, inserting the result into the attribute on behalf of the parser component. In Spirit.Karma the sequence extracts the next container element on behalf of the generator component and passing it the extracted value.

Header

```cpp
#include <boost/spirit/home/support/handles_container.hpp>
```

Also, see Include Structure.

Note

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::traits</td>
</tr>
</tbody>
</table>
Synopsis

```cpp
template <
    typename Component, typename Attribute, typename Context,
    typename Iterator, typename Enable>
struct handles_container
{
    <unspecified>;
};
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>The component type Component which needs to be tested whether it handles</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>container attributes directly.</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>The attribute type Attribute as passed to the sequence operator.</td>
<td>none</td>
</tr>
<tr>
<td>Context</td>
<td>This is the type of the current component execution context.</td>
<td>unused_type</td>
</tr>
<tr>
<td>Iterator</td>
<td>The type, Iterator is the type of the iterators used to invoke the</td>
<td>unused_type</td>
</tr>
<tr>
<td></td>
<td>component.</td>
<td></td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain</td>
<td>void</td>
</tr>
<tr>
<td></td>
<td>specializations of is_container utilizing SFINAE (i.e. boost::enable_if or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>boost::disable_if).</td>
<td></td>
</tr>
</tbody>
</table>

Notation

Component A component type to be tested whether it directly handles container attributes in the context of sequences.

Attribute A container attribute type as passed to the sequence.

T1, T2, ... Arbitrary types

Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>handles_container&lt;Component, Attribute&gt;::type</td>
<td>Result of the metafunction that evaluates to mpl::true_ if a given</td>
</tr>
<tr>
<td></td>
<td>component type Component, handles container attributes directly, mpl::false_ otherwise. Generally, any implementation of handles_container needs to behave as if it was a</td>
</tr>
<tr>
<td></td>
<td>MPL Boolean Constant.</td>
</tr>
</tbody>
</table>

Predefined Specializations

Spirit predefined specializations of this customization point for several types. The following table lists those types together with the conditions for which the corresponding specializations will evaluate to mpl::true_ (see MPL Boolean Constant):
### Template Parameters

<table>
<thead>
<tr>
<th>Component, Attribute</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mpl::false_</strong> (the default).</td>
<td></td>
</tr>
</tbody>
</table>

| rule<Iterator, T1, T2, T3, T4>, Attribute | Returns is_container<A>, where A is the attribute exposed by the rule (Spirit.Qi and Spirit.Karma). |
| grammar<Iterator, T1, T2, T3, T4>, Attribute | Returns is_container<A>, where A is the attribute exposed by the grammar (Spirit.Qi and Spirit.Karma). |

### When to implement

The customization point handles_container needs to be implemented for a specific type whenever this type directly handles container attributes. It is applicable for parsers (Spirit.Qi) and generators (Spirit.Karma). It will have to be implemented under rare circumstances only.

### Transform an Attribute to a Different Type (Qi and Karma)

#### transform_attribute

The template transform_attribute is a type used as an attribute customization point. It is invoked by Qi rule, semantic action and attr_cast, and Karma rule, semantic action and attr_cast. It is used to automatically transform the user provided attribute to the attribute type expected by the right hand side component (for rule), the semantic action, or the embedded component (for attr_cast).

#### Note

The interface of this customization point has been changed with Boost V1.44. We added the Domain template parameter to allow for more fine grained specializations for Spirit.Qi and Spirit.Karma.

### Module Headers

```cpp
#include <boost/spirit/home/support/attributes.hpp>
```

### Notes

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

### Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::traits</td>
</tr>
</tbody>
</table>
Synopsis

template <typename Exposed, typename Transformed, typename Domain, typename Enable>
struct transform_attribute
{
    typedef <unspecified> type;

    static type pre(Exposed& val);
    static void post(Exposed& val, type attr); // Qi only
    static void fail(Exposed&); // Qi only
};

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exposed</td>
<td>The attribute type supplied to the component which needs to be transformed.</td>
<td>none</td>
</tr>
<tr>
<td>Transformed</td>
<td>The attribute type expected by the component to be provided as the result of the transformation.</td>
<td>none</td>
</tr>
<tr>
<td>Domain</td>
<td>The domain of the sub library the template is instantiated in. Typically this is either qi::domain or karma::domain.</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of transform_attribute utilizing SFINAE (i.e. boost::enable_if or boost::disable_if).</td>
<td>void</td>
</tr>
</tbody>
</table>

Notation

Exposed The type, Exposed is the type of the attribute as passed in by the user.

Transformed The type, Transformed is the type of the attribute as passed along to the right hand side of the rule (embedded component of attr_cast).

Domain The domain of the sub library the template is instantiated in. Typically this is either qi::domain or karma::domain.

exposed An instance of type Exposed.

transformed An instance of type Transformed.
### Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transform_attribute&lt;Exposed, Transformed, Domain&gt;::type</code></td>
<td>Evaluates to the type to be used as the result of the transformation (to be passed to the right hand side of the rule or to the embedded component of the <code>attr_cast</code>). Most of the time this is equal to <code>Transformed</code>, but in other cases this might evaluate to <code>Transformed&amp;</code> instead avoiding superfluous object creation.</td>
</tr>
<tr>
<td><code>void transform_attribute&lt;Exposed, Transformed, Domain&gt;::pre(exposed)</code></td>
<td>Do pre-transformation before invoking the right hand side component for rule (or the embedded component for <code>attr_cast</code>). This takes the attribute supplied as by the user (of type <code>Exposed</code>) and returns the attribute to be passed down the component hierarchy (of the type as exposed by the metafunction <code>type</code>). This function will be called in <code>Qi</code> and for <code>Karma</code>.</td>
</tr>
<tr>
<td><code>void transform_attribute&lt;Exposed, Transformed, Domain&gt;::post(exposed, transformed)</code></td>
<td>Do post-transformation after the invocation of the right hand side component for rule (or the embedded component for <code>attr_cast</code>). This takes the original attribute as supplied by the user and the attribute as returned from the right hand side (embedded) component and is expected to propagate the result back into the supplied attribute instance. This function will be called in <code>Qi</code> only.</td>
</tr>
<tr>
<td><code>void transform_attribute&lt;Exposed, Transformed, Domain&gt;::fail(exposed)</code></td>
<td>Handling failing parse operations of the right hand side component for rule (or the embedded component for <code>attr_cast</code>). This function will be called in <code>Qi</code> only.</td>
</tr>
</tbody>
</table>
## Predefined Specializations

<table>
<thead>
<tr>
<th>Template parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exposed, Transformed</td>
<td>type evaluates to Transformed, pre() returns a new instance of Transformed constructed from the argument of type Exposed, post() assigns transformed to exposed.</td>
</tr>
<tr>
<td>optional&lt;Exposed&gt;, Transformed, typename disable_if&lt;is_same&lt;optional&lt;Exposed&gt;, Transformed&gt;::type&gt;</td>
<td>type evaluates to Transformed&amp;, pre() returns a reference to the instance of Transformed stored in the passed optional (the argument of type optional&lt;Exposed&gt;), the optional instance is initialized, if needed. post() does nothing. fail() resets the optional (its parameter) instance to the non-initialized state.</td>
</tr>
<tr>
<td>Exposed&amp;, Transformed</td>
<td>type evaluates to Transformed&amp;, pre() returns a new instance of Transformed constructed from the argument of type Exposed, post() assigns transformed to exposed.</td>
</tr>
<tr>
<td>Attrib&amp;, Attrib</td>
<td>type evaluates to Attrib&amp;, pre() returns it’s argument, post() does nothing.</td>
</tr>
<tr>
<td>Exposed const, Transformed</td>
<td>(usind in Karma only) type evaluates to Transformed, pre() returns it’s argument, post() is not implemented.</td>
</tr>
<tr>
<td>Attrib const&amp;, Attrib</td>
<td>(usind in Karma only) type evaluates to Attrib const&amp;, pre() returns it’s argument, post() is not implemented.</td>
</tr>
<tr>
<td>Attrib const, Attrib</td>
<td>(usind in Karma only) type evaluates to Attrib const&amp;, pre() returns it’s argument, post() is not implemented.</td>
</tr>
<tr>
<td>unused_type, Attrib</td>
<td>type evaluates to unused_type, pre() and post() do nothing.</td>
</tr>
<tr>
<td>Attrib, unused_type</td>
<td>type evaluates to unused_type, pre() and post() do nothing.</td>
</tr>
</tbody>
</table>

### When to implement

The customization point transform_attribute needs to be implemented for a specific pair of types whenever the attribute type supplied to a rule or attr_cast cannot automatically transformed to the attribute type expected by the right hand side of the rule (embedded component of the attr_cast) because the default implementation as shown above is not applicable. Examples for this could be that the type Transformed is not constructible from the type Exposed.

### Store a Parsed Attribute Value (Qi)

After parsing input and generating an attribute value this value needs to assigned to the attribute instance provided by the user. The customization points assign_to_attribute_from_iterators and assign_to_attribute_from_value are utilized to adapt this assignment to the concrete type to be assigned. This section describes both.

### Store an Attribute after a Parser Produced a Pair of Iterators (Qi)

**assign_to_attribute_from_iterators**

The template assign_to_attribute_from_iterators is a type used as an attribute customization point. It is invoked by the those Qi parsers not producing any attribute value but returning a pair of iterators pointing to the matched input sequence. It is used to either store the iterator pair into the attribute instance provided by the user or to convert the iterator pair into an attribute as provided by the user.
Module Headers

```cpp
#include <boost/spirit/home/qi/detail/assign_to.hpp>
```

Also, see Include Structure.

Note

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

Namespace

Name

`boost::spirit::traits`

Synopsis

```cpp
template <typename Attrib, typename Iterator, typename Enable>
struct assign_to_attribute_from_iterators
{
    static void call(Iterator const& first, Iterator const& last, Attrib& attr);
};
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attrib</td>
<td>The type, <code>Attrib</code> is the type of the attribute as passed in by the user.</td>
<td>none</td>
</tr>
<tr>
<td>Iterator</td>
<td>The type, <code>Iterator</code> is the type of the iterators as produced by the parser.</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of <code>assign_to_attribute_from_value</code> utilizing SFINAE (i.e. <code>boost::enable_if</code> or <code>boost::disable_if</code>).</td>
<td>void</td>
</tr>
</tbody>
</table>

Notation

- **Attrib**: A type to be used as the target to store the attribute value in.
- **attr**: An attribute instance of type `Attrib`.
- **Iterator**: The iterator type used by the parser. This type usually corresponds to the iterators as passed in by the user.
- **begin, end**: Iterator instances of type `Iterator` pointing to the begin and the end of the matched input sequence.
### Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assign_to_attribute_from_iterators&lt;Attrib, Iterator&gt;::call(b, e, attr)</code></td>
<td>Use the iterators <code>begin</code> and <code>end</code> to initialize the attribute <code>attr</code>.</td>
</tr>
</tbody>
</table>

### Predefined Specializations

<table>
<thead>
<tr>
<th>Template Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Attrib, Iterator</code></td>
<td>Execute an assignment <code>attr = Attrib(begin, end)</code>.</td>
</tr>
<tr>
<td><code>unused_type, T</code></td>
<td>Do nothing.</td>
</tr>
</tbody>
</table>

### When to implement

The customization point `assign_to_attribute_from_iterators` needs to be implemented for a specific type whenever the default implementation as shown above is not applicable. Examples for this could be that the type `Attrib` is not constructible from the pair of iterators.

### Store an Attribute Value after a Parser Produced a Value (Qi)

#### `assign_to_attribute_from_value`

The template `assign_to_attribute_from_value` is a type used as an attribute customization point. It is invoked by all primitive `Qi` parsers in order to store a parsed attribute value into the attribute instance provided by the user, if this attribute is not a container type (`is_container<T>::type` evaluates to `mpl::false_`, where `T` is the attribute type).

### Module Headers

```
#include <boost/spirit/home/qi/detail/assign_to.hpp>
```

Also, see Include Structure.

### Note

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

### Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boost::spirit::traits</code></td>
</tr>
</tbody>
</table>
Synopsis

```cpp
template <typename Attrib, typename T, typename Enable>
struct assign_to_attribute_from_value
{
    static void call(T const& val, Attrib& attr);
};
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attrib</td>
<td>The type, <code>Attrib</code> is the type of the attribute as passed in by the user. This type is not a container type (<code>is_container&lt;Attrib&gt;::type</code> evaluates to <code>mpl::false_</code>).</td>
<td>none</td>
</tr>
<tr>
<td>T</td>
<td>The type, <code>T</code> is the type of the attribute instance as produced by the parser.</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of <code>assign_to_attribute_from_value</code> utilizing SFINAE (i.e. <code>boost::enable_if</code> or <code>boost::disable_if</code>).</td>
<td>void</td>
</tr>
</tbody>
</table>

Notation

- `Attrib` A type to be used as the target to store the attribute value in. This type is guaranteed not to be a container type (`is_container<Attrib>::type` evaluates to `mpl::false_`).
- `attr` An attribute instance of type `Attrib`.
- `T` A type as produced by the parser. The parser temporarily stores its parsed values using this type.
- `t` An attribute instance of type `T`.

Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assign_to_attribute_from_value&lt;Attrib,T&gt;::call(t, attr)</code></td>
<td>Copy (assign) the value <code>t</code> to the attribute <code>attr</code>.</td>
</tr>
</tbody>
</table>

Predefined Specializations

<table>
<thead>
<tr>
<th>Template Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attrib, T</td>
<td>Assign the argument <code>t</code> to <code>attr</code>.</td>
</tr>
<tr>
<td>unused_type, T</td>
<td>Do nothing.</td>
</tr>
</tbody>
</table>
When to implement

The customization point `assign_to_attribute_from_value` needs to be implemented for a specific type whenever the default implementation as shown above is not applicable. Examples for this could be that the type `Attrib` is not copy constructible.

Store an Attribute Value into a Container after a Parser Produced a Value (Qi)

`assign_to_container_from_value`

The template `assign_to_container_from_value` is a type used as an attribute customization point. It is invoked by all primitive `Qi` parsers in order to store a parsed attribute value into the attribute instance provided by the user, if this attribute is a container type (`is_container<T>::type` evaluates to `mpl::true_`, where `T` is the attribute type).

Module Headers

```cpp
#include <boost/spirit/home/qi/detail/assign_to.hpp>
```

Also, see Include Structure.

Note

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::traits</td>
</tr>
</tbody>
</table>
Synopsis

```cpp
template <typename Attrib, typename T, typename Enable>
struct assign_to_container_from_value
{
    static void call(T const& val, Attrib& attr);
};
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attrib</td>
<td>The type, Attrib is the type of the attribute as passed in by the user. This type is a container type (is_container&lt;Attrib&gt;::type evaluates to mpl::true_).</td>
<td>none</td>
</tr>
<tr>
<td>T</td>
<td>The type, T is the type of the attribute instance as produced by the parser.</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of assign_to_container_from_value utilizing SFINAE (i.e. boost::enable_if or boost::disable_if).</td>
<td>void</td>
</tr>
</tbody>
</table>

Notation

- **Attrib**: A type to be used as the target to store the attribute value in. This type is guaranteed to be a container type (is_container<Attrib>::type evaluates to mpl::true_).
- **attr**: An attribute instance of type Attrib.
- **T**: A type as produced by the parser. The parser temporarily stores its parsed values using this type.
- **t**: An attribute instance of type T.

Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign_to_container_from_value&lt;Attrib, T&gt;::call(t, attr)</td>
<td>Add the value t to the container attribute attr.</td>
</tr>
</tbody>
</table>

Predefined Specializations

<table>
<thead>
<tr>
<th>Template Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attrib, T</td>
<td>Add the argument t to attr.</td>
</tr>
<tr>
<td>unused_type, T</td>
<td>Do nothing.</td>
</tr>
</tbody>
</table>
When to implement

The customization point `assign_to_container_from_value` needs to be implemented for a specific type whenever the default implementation as shown above is not applicable. Examples for this could be that the type `Attrib` is not copy constructible.

Store Parsed Attribute Values into a Container (Qi)

In order to customize Spirit to accept a given data type as a container for elements parsed by any of the repetitive parsers (`Kleene`, `Plus`, `List`, and `Repeat`) two attribute customization points have to be specialized: `traits::container_value` and `traits::push_back_container`. This section describes both.

Determine the Type to be Stored in a Container (Qi)

`container_value`

The template `container_value` is a template meta function used as an attribute customization point. It is invoked by the `Qi` repetitive parsers (`Kleene`, `Plus`, `List`, and `Repeat`) to determine the type to store in a container.

Module Headers

```cpp
#include <boost/spirit/home/support/container.hpp>
```

Also, see Include Structure.

Note

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

Namespace

Name

```
boost::spirit::traits
```
Synopsis

```cpp
template <typename Container, typename Enable>
struct container_value
{
    typedef <unspecified> type;
};
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td>The type Container is the type for which the type of the elements has to be deduced.</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of <code>container_value</code> utilizing SFINAE (i.e. <code>boost::enable_if</code> or <code>boost::disable_if</code>).</td>
<td>void</td>
</tr>
</tbody>
</table>

`C` A type to be tested whether it needs to be treated as a container.

`T1, T2, ...` Arbitrary types

Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>container_value&lt;C&gt;::type</code></td>
<td>Metafunction that evaluates to the type to be stored in a given container type, C.</td>
</tr>
</tbody>
</table>

Predefined Specializations

Spirit predefines specializations of this customization point for several types. The following table lists those types together with the types exposed and the corresponding semantics:

<table>
<thead>
<tr>
<th>Template Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>C</code></td>
<td>The non-const <code>value_type</code> of the given container type, C.</td>
</tr>
<tr>
<td><code>boost::optional&lt;C&gt;</code></td>
<td>Returns <code>container_value&lt;C&gt;::type</code></td>
</tr>
<tr>
<td><code>boost::variant&lt;T1, T2, ...&gt;</code></td>
<td>Returns <code>container_value&lt;TN&gt;::value</code> for the first TN (out of T1, T2, ...) for which <code>is_container&lt;TN&gt;::type</code> evaluates to <code>mpl::true_</code>. Otherwise it will return <code>unused_type</code>.</td>
</tr>
<tr>
<td><code>unused_type</code></td>
<td>Returns <code>unused_type</code>.</td>
</tr>
</tbody>
</table>

When to implement

The customization point `is_container` needs to be implemented for a specific type whenever this type is to be used as an attribute in place of a STL container. It is applicable for parsers (Spirit:Qi) only. As a rule of thumb: it has to be implemented whenever a certain type is to be passed as an attribute to a parser normally exposing a STL container and if the type does not expose the interface of a STL container (i.e. no embedded typedef for `value_type`). These components have an attribute propagation rule in the form:
a: A --> Op(a): vector<A>

where Op(a) stands for any meaningful operation on the component a.

**Related Attribute Customization Points**

If this customization point is implemented, the following other customization points might need to be implemented as well.

<table>
<thead>
<tr>
<th>Name</th>
<th>When to implement</th>
</tr>
</thead>
<tbody>
<tr>
<td>traits::push_back_container</td>
<td>Qi: List, Kleene, Plus, Repeat.</td>
</tr>
<tr>
<td>traits::clear_value</td>
<td>Qi: List, Kleene, Plus, Repeat.</td>
</tr>
</tbody>
</table>

**Example**

Here is an example showing the default implementation of the `traits::container_value` customization point provided by the library:

```cpp
template <typename Container, typename Enable/* = void*/>
struct container_value
: detail::remove_value_const<typename Container::value_type>
{};
```

This template is instantiated by the library at the appropriate places while using the supplied container type as the template argument. The embedded `type` is used as the attribute type while parsing the elements to be store in that container.

The following example shows the predefined specialization for `unused_type`:

```cpp
template <>
struct container_value<unused_type>
{
    typedef unused_type type;
};
```

which defines its embedded `type` to be `unused_type` as well, this way propagating the 'don't care' attribute status to the embedded parser.

**Store a Parsed Attribute Value into a Container (Qi)**

**push_back_container**

The template `push_back_container` is a type used as an attribute customization point. It is invoked by the `Qi` repetitive parsers (Kleene, Plus, List, and Repeat) to store a parsed attribute value into a container.

**Module Headers**

```cpp
#include <boost/spirit/home/support/container.hpp>
```

Also, see Include Structure.

**Note**

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.
Namespace

Name
boost::spirit::traits

Synopsis

```cpp
template <typename Container, typename Attrib, typename Enable>
struct push_back_container
{
    static bool call(Container& c, Attrib const& val);
};
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td>The type, <code>Container</code> needs to be tested whether it has to be treated as a container.</td>
<td>none</td>
</tr>
<tr>
<td>Attrib</td>
<td>The type, <code>Attrib</code> is the one returned from the customization point <code>traits::container_value</code> and represents the attribute value to be stored in the container of type <code>Container</code>.</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of <code>push_back_container</code> utilizing SFINAE (i.e. <code>boost::enable_if</code> or <code>boost::disable_if</code>).</td>
<td>void</td>
</tr>
</tbody>
</table>

Notation

- `C` A type to be used as a container to store attribute values in.
- `c` A container instance of type `C`.
- `Attrib` A type to be used as a container to store attribute values in.
- `attr` An attribute instance of type `Attrib`.
- `T1, T2, ...` Arbitrary types

Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>push_back_container&lt;C, Attrib&gt;::call(c, attr)</code></td>
<td>Static function that is invoked whenever an attribute value, <code>attr</code> needs to be stored into the container instance <code>c</code>. This function should return <code>true</code> on success and <code>false</code> otherwise. Returning false causes the corresponding parser to fail.</td>
</tr>
</tbody>
</table>
Predefined Specializations

Spirit predefines specializations of this customization point for several types. The following table lists those types together with the types exposed and the corresponding semantics:

<table>
<thead>
<tr>
<th>Template Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, Attrib</td>
<td>Store the provided attribute instance attr into the given container c using the function call c.insert(c.end(), attr).</td>
</tr>
<tr>
<td>boost::optional&lt;C&gt;, Attrib</td>
<td>If the provided instance of boost::optional&lt;&gt; is not initialized, invoke the appropriate initialization and afterwards apply the customization point push_back_container&lt;C, Attrib&gt;, treating the instance held by the optional (of type C) as the container to store the attribute in.</td>
</tr>
<tr>
<td>boost::variant&lt;T1, T2, ...&gt;, Attrib</td>
<td>If the instance of the variant currently holds a value with a type, TN, for which is_container&lt;TN&gt;::type evaluates to mpl::true_, this customization point specialization will apply push_back_container&lt;TN, Attrib&gt;, treating the instance held by the variant (of type TN) as the container to store the attribute in. Otherwise it will raise an assertion.</td>
</tr>
<tr>
<td>unused_type</td>
<td>Do nothing.</td>
</tr>
</tbody>
</table>

When to Implement

The customization point push_back_container needs to be implemented for a specific type whenever this type is to be used as an attribute in place of a STL container. It is applicable for parsers (Spirit.Qi) only. As a rule of thumb: it has to be implemented whenever a certain type is to be passed as an attribute to a parser normally exposing a STL container and if the type does not expose the interface of a STL container (i.e. no function being equivalent to c.insert(c.end(), attr). These components have an attribute propagation rule in the form:

\[
\text{a: A} \rightarrow \text{Op(a): vector\text{<A>}}
\]

where \(\text{Op(a)}\) stands for any meaningful operation on the component a.

Related Attribute Customization Points

If this customization point is implemented, the following other customization points might need to be implemented as well.

<table>
<thead>
<tr>
<th>Name</th>
<th>When to implement</th>
</tr>
</thead>
<tbody>
<tr>
<td>traits::container_value</td>
<td>Qi: List, Kleene, Plus, Repeat.</td>
</tr>
<tr>
<td>traits::clear_value</td>
<td>Qi: List, Kleene, Plus, Repeat.</td>
</tr>
</tbody>
</table>

Example

Here is an example showing the default implementation of the traits::container_value customization point provided by the library:
template <typename Container, typename T, typename Enable/* = void*/>
struct push_back_container
{
    static bool call(Container& c, T const& val)
    {
        c.insert(c.end(), val);
        return true;
    }
};

This template is instantiated by the library at the appropriate places while using the supplied container and element types as the template arguments. The member function call() will be called whenever an element has to be added to the supplied container.

The following example shows the predefined specialization for unused_type:

```cpp
template <typename Container>
bool push_back(Container&, unused_type)
{
    return true;
}
```

which defines an empty member function call().

**Re-Initialize an Attribute Value before Parsing (Qi)**

**clear_value**

The template clear_value is a type used as an attribute customization point. It is invoked by the Qi repetitive parsers (Kleene, Plus, List, and Repeat) in order to re-initialize the attribute instance passed to the embedded parser after it has been stored in the provided container. This re-initialized attribute instance is reused during the next iteration of the repetitive parser.

**Module Headers**

```
#include <boost/spirit/home/support/attributes.hpp>
```

Also, see Include Structure.

**Note**

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

**Namespace**

**Name**

```
boost::spirit::traits
```
Synopsis

\begin{verbatim}
template <typename Attrib, typename Enable>
struct clear_value
{
    static void call(Attrib& val);
};
\end{verbatim}

**Template parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attrib</td>
<td>The type, Attrib of the attribute to be re-initialized.</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of clear_value utilizing SFINAE (i.e. boost::enable_if or boost::disable_if).</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notation**

Attrib A type to be used as a container to store attribute values in.
attr An attribute instance of type Attrib.
T1, T2,... Arbitrary types

**Expression Semantics**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear_value&lt;Attrib&gt;::call(Attrib&amp; attr)</td>
<td>Re-initialize the instance referred to by attr in the most efficient way.</td>
</tr>
</tbody>
</table>

**Predefined Specializations**

Spirit predefines specializations of this customization point for several types. The following table lists those types together with the types exposed and the corresponding semantics:
### Template Parameters

<table>
<thead>
<tr>
<th>Attrib</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re-initialize using assignment of default constructed value.</td>
</tr>
</tbody>
</table>

| Any type T for which is_container>::mpl::true_ |
| Call the member function attr.clear() for the passed attribute instance. |

| boost::optional<Attrib> |
| Clear the optional instance and leave it uninitialized. |

| boost::variant<T1, T2, ...> |
| Invoke the clear_value customization point for the currently held value. |

| fusion::tuple<T1, T2, ...> |
| Invoke the clear_value customization point for all elements of the tuple. |

| unused_type |
| Do nothing. |

### When to Implement

The customization point clear_value needs to be implemented for a specific type whenever this type is to be used as an attribute to be stored into a STL container and if the type cannot be re-initialized using one of the specializations listed above. Examples for this might be types not being default constructible or container types not exposing a member function clear().

### Extract an Attribute Value to Generate Output (Karma)

**extract_from**

Before generating output for a value this value needs to extracted from the attribute instance provided by the user. The customization point extract_from_attribute is utilized to adapt this extraction for any data type possibly used to store the values to output.

**Note**

The interface of this customization point has been changed with Boost V1.44. We added the Exposed template parameter to allow for more fine grained specializations of the required Spirit.Karma attribute transformations.

### Module Headers

```cpp
#include <boost/spirit/home/karma/detail/extract_from.hpp>
```

Also, see Include Structure.

**Note**

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

### Namespace

| Name |
| boost::spirit::traits |
**Synopsis**

```cpp
template <typename Exposed, typename Attrib, typename Enable>
struct extract_from_attribute
{
    typedef <unspecified> type;
    template <typename Context>
    static type call(Attrib const& attr, Context& context);
};
```

**Template parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exposed</td>
<td>The type, Exposed of the attribute natively exposed by the component the extract_from_attribute is invoked from.</td>
<td>none</td>
</tr>
<tr>
<td>Attrib</td>
<td>The type, Attrib of the attribute to be used to generate output from.</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of clear_value utilizing SFINAE (i.e. boost::enable_if or boost::disable_if).</td>
<td>void</td>
</tr>
<tr>
<td>Context</td>
<td>This is the type of the current generator execution context.</td>
<td></td>
</tr>
</tbody>
</table>

**Notation**

- Exposed: A type exposed as the native attribute of a component.
- Attrib: A type to be used to generate output from.
- attr: An attribute instance of type Attrib.
- ctx: An instance of type Context.

**Expression Semantics**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract_from_attribute&lt;Exposed, Attrib&gt;::call(attr, ctx)</td>
<td>Extract the value to generate output from attr and return it to the caller.</td>
</tr>
</tbody>
</table>

**Predefined Specializations**

*Spirit* predefines specializations of this customization point for several types. The following table lists those types together with the types exposed and the corresponding semantics:
### Template Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Attrib</code></td>
<td>The exposed typedef <code>type</code> is defined to <code>Attrib const&amp;</code>. The function <code>call()</code> returns the argument by reference without change.</td>
</tr>
<tr>
<td><code>boost::optional&lt;Attrib&gt;</code></td>
<td>The exposed typedef <code>type</code> is defined to <code>Attrib const&amp;</code>. The function <code>call()</code> returns the value held by the <code>optional&lt;&gt;</code> argument by reference without change.</td>
</tr>
<tr>
<td><code>boost::reference_wrapper&lt;Attrib&gt;</code></td>
<td>The exposed typedef <code>type</code> is defined to <code>Attrib const&amp;</code>. The function <code>call()</code> returns the value held by the <code>reference_wrapper&lt;&gt;</code> argument by reference without change.</td>
</tr>
<tr>
<td><code>unused_type</code></td>
<td>The exposed typedef <code>type</code> is defined to <code>unused_type</code>. The function <code>call()</code> returns an instance of <code>unused_type</code>.</td>
</tr>
</tbody>
</table>

### When to implement

The customization point `extract_from_attribute` needs to be implemented for a specific type whenever the default implementation as shown above is not applicable. Examples for this could be that the type to be extracted is different from `Attrib` and is not copy constructible.

### Extract From a Container Attribute Value to Generate Output (Karma)

**extract_from_container**

Before generating output for a value this value needs to extracted from the attribute instance provided by the user. The customization point `extract_from_container` is utilized to adapt this extraction for any data type possibly used to store the values to output.

#### Note

The interface of this customization point has been changed with Boost V1.44. We added the Exposed template parameter to allow for more fine grained specializations of the required `Spirit.Karma` attribute transformations.

### Module Headers

```cpp
#include <boost/spirit/home/karma/detail/extract_from.hpp>
```

Also, see Include Structure.

#### Note

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

### Namespace

**Name**

`boost::spirit::traits`
**Synopsis**

```cpp
template <typename Exposed, typename Attrib, typename Enable>
struct extract_from_container
{
    typedef <unspecified> type;
    template <typename Context>
    static type call(Attrib const& attr, Context& context);
};
```

**Template parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exposed</td>
<td>The type, Exposed of the attribute natively exposed by the component the <code>extract_from_container</code> is invoked from.</td>
<td>none</td>
</tr>
<tr>
<td>Attrib</td>
<td>The type, Attrib is the container attribute to be used to generate output from.</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of <code>clear_value</code> utilizing SFINAE (i.e. <code>boost::enable_if</code> or <code>boost::disable_if</code>).</td>
<td>void</td>
</tr>
<tr>
<td>Context</td>
<td>This is the type of the current generator execution context.</td>
<td></td>
</tr>
</tbody>
</table>

**Notation**

- **Exposed**: A type exposed as the native attribute of a component.
- **Attrib**: A container type to be used to generate output from.
- **attr**: An attribute instance of type `Attrib`.
- **ctx**: An instance of type `Context`.

**Expression Semantics**

```
extract_from_container<Exposed, Attrib>::call(attr, ctx)
```

Extract the value to generate output from the container given by `attr` and return it to the caller.

**Predefined Specializations**

`Spirit` predefines specializations of this customization point for several types. The following table lists those types together with the types exposed and the corresponding semantics:
## Template Parameters

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attrib</td>
<td>The exposed typedef type is defined to Attrib const&amp;. The function call() returns the argument by reference without change.</td>
</tr>
<tr>
<td>unused_type</td>
<td>The exposed typedef type is defined to unused_type. The function call() returns an instance of unused_type.</td>
</tr>
</tbody>
</table>

### When to implement

The customization point `extract_from_container` needs to be implemented for a specific container type whenever the default implementation as shown above is not applicable. Examples for this could be that the type to be extracted is different from `Attrib` and is not copy constructible.

#### Example

TBD

---

### Extract Attribute Values to Generate Output from a Container (Karma)

#### Determine the Type of the Iterator of a Container

The template `container_iterator` is a template meta-function used as an attribute customization point. It is invoked by the `Karma` repetitive generators (such as `List (?)`, `Kleene (unary *)`, `Plus (unary +)`, and `Repeat`) in order to determine the type of the iterator to use to iterate over the items to be exposed as the elements of a container.

#### Module Headers

```
#include <boost/spirit/home/support/container.hpp>
```

Also, see Include Structure.

---

### Note

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

#### Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::traits</td>
</tr>
</tbody>
</table>
Synopsis

template <typename Container, typename Enable>
struct container_iterator
{
    typedef <unspecified> type;
};

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td>The type, Container for which the iterator type has to be returned</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of container_iterator utilizing SFINAE (i.e. boost::enable_if or boost::disable_if).</td>
<td>void</td>
</tr>
</tbody>
</table>

Notation

C A container type the iterator type needs to be evaluated for.

Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>container_iterator&lt;C&gt;::type</td>
<td>Result of the metafunction that evaluates the type to be used as the iterator for accessing all elements of a container, C.</td>
</tr>
</tbody>
</table>

The returned type conceptually needs to be equivalent to a standard forward iterator. But it does not have to expose the standardized interface. If this customization point is implemented for a certain container type, all related customization points need to be implemented as well (see Related Attribute Customization Points below). This encapsulates the specific iterator interface required for a given type. The minimal requirements for a type to be exposed as an iterator in this context are:

- it needs to be comparable for equality (see traits::compare_iterators),
- it needs to be incrementable (see traits::next_iterator),
- it needs to be dereferencible (see traits::deref_iterator).

Predefined Specializations

Spirit predefines specializations of this customization point for several types. The following table lists those types together with the types returned by the embedded typedef type:

<table>
<thead>
<tr>
<th>Template Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Returns C::iterator.</td>
</tr>
<tr>
<td>C const</td>
<td>Returns C::const_iterator.</td>
</tr>
<tr>
<td>unused_type</td>
<td>Returns unused_type const*.</td>
</tr>
</tbody>
</table>
**When to implement**

The customization point `container_iterator` needs to be implemented for a specific type whenever this type is to be used as an attribute in place of a STL container. It is applicable for generators (*Spirit.Karma*) only. As a rule of thumb: it has to be implemented whenever a certain type is to be passed as an attribute to a generator normally exposing a STL container, `C` and if the type does not expose the interface of a STL container (i.e. `is_container<C>::type` would normally return `mpl::false_`).

**Related Attribute Customization Points**

If this customization point is implemented, the following other customization points might need to be implemented as well.

<table>
<thead>
<tr>
<th>Name</th>
<th>When to implement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>traits::is_container</code></td>
<td>Needs to be implemented whenever a type is to be used as a container attribute in <em>Karma</em>.</td>
</tr>
<tr>
<td><code>traits::container_iterator</code></td>
<td>*Karma: List (%), Kleene (unary <em>), Plus (unary +), Repeat.</em></td>
</tr>
<tr>
<td><code>traits::begin_container</code></td>
<td>*Karma: List (%), Kleene (unary <em>), Plus (unary +), Repeat.</em></td>
</tr>
<tr>
<td><code>traits::end_container</code></td>
<td>*Karma: List (%), Kleene (unary <em>), Plus (unary +), Repeat.</em></td>
</tr>
<tr>
<td><code>traits::dereference_iterator</code></td>
<td>*Karma: List (%), Kleene (unary <em>), Plus (unary +), Repeat.</em></td>
</tr>
<tr>
<td><code>traits::next_iterator</code></td>
<td>*Karma: List (%), Kleene (unary <em>), Plus (unary +), Repeat.</em></td>
</tr>
<tr>
<td><code>traits::compare_iterators</code></td>
<td>*Karma: List (%), Kleene (unary <em>), Plus (unary +), Repeat.</em></td>
</tr>
</tbody>
</table>

**Example**

Here are the header files needed to make the example code below compile:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <iostream>
#include <vector>
```

The example (for the full source code please see here: customize_embedded_container.cpp) uses the data structure

```cpp
namespace client
{
    struct embedded_container
    {
        // expose the iterator of the embedded vector as our iterator
        typedef std::vector<int>::const_iterator iterator;

        // expose the type of the held data elements as our type
        typedef std::vector<int>::value_type type;

        // this is the vector holding the actual elements we need to generate
        // output from
        std::vector<int> data;
    };
}
```

as a direct container attribute to the *List (\%)* generator. In order to make this data structure compatible we need to specialize a couple of attribute customization points: `traits::is_container`, `traits::container_iterator`, `traits::begin_container`, `traits::end_container`, and `traits::dereference_iterator`. As you can see the specializations simply expose the embedded `std::vector<int>` as the container to use. We don't need to specialize the customization points related to iterators `traits::dereference_iterator`, `traits::next_iterator`, and `traits::compare_iterators`.
traits::next_iterator, and traits::compare_iterators) as we expose a standard iterator and the default implementation of these customizations handles standard iterators out of the box.

```cpp
namespace boost { namespace spirit { namespace traits {

// The specialization of the template 'is_container<>' will tell the
// library to treat the type 'client::embedded_container' as a
// container holding the items to generate output from.
template <>
struct is_container<client::embedded_container const>
  : mpl::true_
{};

// The specialization of the template 'container_iterator<>' will be
// invoked by the library to evaluate the iterator type to be used
// for iterating the data elements in the container. We simply return
// the type of the iterator exposed by the embedded 'std::vector<int>'.
template <>
struct container_iterator<client::embedded_container const>
{
  typedef client::embedded_container::iterator type;
};

// The specialization of the templates 'begin_container<>' and
// 'end_container<>' below will be used by the library to get the iterators
// pointing to the begin and the end of the data to generate output from.
// These specializations simply return the 'begin' and 'end' iterators as
// exposed by the embedded 'std::vector<int>'.
// The passed argument refers to the attribute instance passed to the list
generator.
template <>
struct begin_container<client::embedded_container const>
{
  static client::embedded_container::iterator
call(client::embedded_container const& d)
  {
    return d.data.begin();
  }
};

template <>
struct end_container<client::embedded_container const>
{
  static client::embedded_container::iterator
call(client::embedded_container const& d)
  {
    return d.data.end();
  }
};
}}}
```

The last code snippet shows an example using an instance of the data structure client::embedded_container to generate output from a List ($) generator:
As you can see, the specializations for the customization points as defined above enable the seamless integration of the custom data structure without having to modify the output format or the generator itself.

For other examples of how to use the customization point `container_iterator` please see here: `use_as_container` and `counter_example`.

### Get the Iterator pointing to the Begin of a Container Attribute

**begin_container**

The template `begin_container` is a type used as an attribute customization point. It is invoked by the *Karma* repetitive generators (such as `List(%)`, `Kleene (unary *)`, `Plus (unary +)`, and `Repeat`) in order to get an iterator pointing to the first element of the container holding the attributes to generate output from.

**Module Headers**

```cpp
#include <boost/spirit/home/support/container.hpp>
```

Also, see [Include Structure](#).

**Note**

This header file does not need to be included directly by any user program as it is normally included by other *Spirit* header files relying on its content.

**Namespace**

**Name**

`boost::spirit::traits`
Synopsis

```
template <typename Container, typename Enable>
struct begin_container
{
    static typename container_iterator<Container>::type
    call(Container& c);
};
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td>The type, <code>Container</code> for which the iterator pointing to the first element has to be returned</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of <code>begin_container</code> utilizing SFINAE (i.e. <code>boost::enable_if</code> or <code>boost::disable_if</code>).</td>
<td>void</td>
</tr>
</tbody>
</table>

Notation

- `C` A container type the begin iterator needs to be returned for.
- `c` An instance of a container, `C`.

Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>begin_container&lt;C&gt;::call(c)</code></td>
<td>Return the iterator usable to dereference the first element of the given container, <code>c</code>. The type of the returned iterator is expected to be the same as the type returned by the customization point <code>traits::container_iterator</code>.</td>
</tr>
</tbody>
</table>

The returned instance conceptually needs to be equivalent to a standard forward iterator. But it does not have to expose the standardized interface. If this customization point is implemented for a certain container type, all related customization points need to be implemented as well (see Related Attribute Customization Points below). This encapsulates the specific iterator interface required for a given type. The minimal requirements for a type to be exposed as an iterator in this context are:

- it needs to be comparable for equality (see `traits::compare_iterators`),
- it needs to be incrementable (see `traits::next_iterator`),
- it needs to be dereferencible (see `traits::dereference_iterator`).

Predefined Specializations

Spirit predefines specializations of this customization point for several types. The following table lists those types together with the types returned by the embedded typedef `type`:

---

460
Table:

<table>
<thead>
<tr>
<th>Template Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Returns c.begin().</td>
</tr>
<tr>
<td>C const</td>
<td>Returns c.begin().</td>
</tr>
<tr>
<td>unused_type</td>
<td>Returns &amp;unused.</td>
</tr>
</tbody>
</table>

**When to implement**

The customization point `begin_container` needs to be implemented for a specific type whenever this type is to be used as an attribute in place of a STL container. It is applicable for generators (`Spirit.Karma`) only. As a rule of thumb: it has to be implemented whenever a certain type is to be passed as an attribute to a generator normally exposing a STL container, `C` and if the type does not expose the interface of a STL container (i.e. `is_container<C>::type` would normally return `mpl::false_`).

**Related Attribute Customization Points**

If this customization point is implemented, the following other customization points might need to be implemented as well.

<table>
<thead>
<tr>
<th>Name</th>
<th>When to implement</th>
</tr>
</thead>
<tbody>
<tr>
<td>traits::is_container</td>
<td>Needs to be implemented whenever a type is to be used as a container attribute in <code>Karma</code>.</td>
</tr>
<tr>
<td>traits::container_iterator</td>
<td>Karma: <code>List (%)</code>, <code>Kleene (unary *)</code>, <code>Plus (unary +)</code>, <code>Repeat</code>.</td>
</tr>
<tr>
<td>traits::begin_container</td>
<td>Karma: <code>List (%)</code>, <code>Kleene (unary *)</code>, <code>Plus (unary +)</code>, <code>Repeat</code>.</td>
</tr>
<tr>
<td>traits::end_container</td>
<td>Karma: <code>List (%)</code>, <code>Kleene (unary *)</code>, <code>Plus (unary +)</code>, <code>Repeat</code>.</td>
</tr>
<tr>
<td>traits::dereff_iterator</td>
<td>Karma: <code>List (%)</code>, <code>Kleene (unary *)</code>, <code>Plus (unary +)</code>, <code>Repeat</code>.</td>
</tr>
<tr>
<td>traits::next_iterator</td>
<td>Karma: <code>List (%)</code>, <code>Kleene (unary *)</code>, <code>Plus (unary +)</code>, <code>Repeat</code>.</td>
</tr>
<tr>
<td>traits::compare_iterators</td>
<td>Karma: <code>List (%)</code>, <code>Kleene (unary *)</code>, <code>Plus (unary +)</code>, <code>Repeat</code>.</td>
</tr>
</tbody>
</table>

**Example**

For examples of how to use the customization point `begin_container` please see here: `embedded_container_example`, `use_as_container`, and `counter_example`.

**Get the Iterator pointing to the End of a Container Attribute**

`end_container`

The template `end_container` is a type used as an attribute customization point. It is invoked by the `Karma` repetitive generators (such as `List (%)`, `Kleene (unary *)`, `Plus (unary +)`, and `Repeat`) in order to get an iterator pointing to the end of the container holding the attributes to generate output from.

**Module Headers**

```
#include <boost/spirit/home/support/container.hpp>
```

Also, see `Include Structure`. 
Note

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

Namespace

Name

boost::spirit::traits

Synopsis

```cpp
template <typename Container, typename Enable>
struct end_container
{
    static typename container_iterator<Container>::type
    call(Container& c);
};
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td>The type, Container for which the iterator pointing to the first element has to be returned</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of end_container utilizing SFINAE (i.e. boost::enable_if or boost::disable_if).</td>
<td>void</td>
</tr>
</tbody>
</table>

Notation

C A container type the end iterator needs to be returned for.

c An instance of a container, C.

Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>end_container&lt;C&gt;::call(c)</td>
<td>Return the iterator usable to compare a different iterator with in order to detect whether the other iterator reached the end of the given container, c. The type of the returned iterator is expected to be the same as the type returned by the customization point traits::container_iterator.</td>
</tr>
</tbody>
</table>

Predefined Specializations

Spirit predefines specializations of this customization point for several types. The following table lists those types together with the types returned by the embedded typedef type:
Template Parameters | Semantics
---|---
C | Returns c.end().
C const | Returns c.end().
unused_type | Returns &unused.

### When to implement

The customization point `end_container` needs to be implemented for a specific type whenever this type is to be used as an attribute in place of a STL container. It is applicable for generators (`Spirit.Karma`) only. As a rule of thumb: it has to be implemented whenever a certain type is to be passed as an attribute to a generator normally exposing a STL container, `C` and if the type does not expose the interface of a STL container (i.e. `is_container<C>::type` would normally return `mpl::false_`).

### Related Attribute Customization Points

If this customization point is implemented, the following other customization points might need to be implemented as well.

<table>
<thead>
<tr>
<th>Name</th>
<th>When to implement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>traits::is_container</code></td>
<td>Needs to be implemented whenever a type is to be used as a container attribute in <code>Karma</code>.</td>
</tr>
<tr>
<td><code>traits::container_iterator</code></td>
<td><code>Karma</code>: List (%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td><code>traits::begin_container</code></td>
<td><code>Karma</code>: List (%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td><code>traits::end_container</code></td>
<td><code>Karma</code>: List (%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td><code>traits::derefer_iterator</code></td>
<td><code>Karma</code>: List (%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td><code>traits::next_iterator</code></td>
<td><code>Karma</code>: List (%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td><code>traits::compare_iterators</code></td>
<td><code>Karma</code>: List (%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
</tbody>
</table>

### Example

For examples of how to use the customization point `end_container` please see here: `embedded_container_example`, `use_as_container`, and `counter_example`.

### Increment the Iterator pointing into a Container Attribute

**next_iterator**

The template `next_iterator` is a type used as an attribute customization point. It is invoked by the `Karma` repetitive generators (such as `List (%)`, `Kleene (unary *)`, `Plus (unary +)`, and `Repeat`) in order to get an iterator pointing to the next element of a container holding the attributes to generate output from.

### Module Headers

```cpp
#include <boost/spirit/home/support/container.hpp>
```

Also, see Include Structure.
Note
This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

Namespace
Name
boost::spirit::traits

Synopsis

```cpp
template <typename Iterator, typename Enable>
struct next_iterator
{
    static void call(Iterator& it);
};
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator</td>
<td>The type, Iterator of the iterator to increment. This is the same as the type returned by the customization point traits::container_iterator.</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of next_iterator utilizing SFINAE (i.e. boost::enable_if or boost::disable_if).</td>
<td>void</td>
</tr>
</tbody>
</table>

Notation

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator</td>
<td>An iterator type.</td>
</tr>
<tr>
<td>it</td>
<td>An instance of an iterator of type Iterator.</td>
</tr>
<tr>
<td>C</td>
<td>A container type whose iterator type is Iterator.</td>
</tr>
</tbody>
</table>

Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_iterator&lt;Iterator&gt;::call(it)</td>
<td>Increment the iterator pointing so that it is pointing to the next element.</td>
</tr>
</tbody>
</table>

Predefined Specializations

Spirit predefined specializations of this customization point for several types. The following table lists those types together with the types returned by the embedded typedef type:
### Template Parameters

<table>
<thead>
<tr>
<th>Iterator</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused_type const*</td>
<td>Does nothing.</td>
</tr>
</tbody>
</table>

### When to implement

The customization point `next_iterator` needs to be implemented for a specific iterator type whenever the container this iterator belongs to is to be used as an attribute in place of a STL container. It is applicable for generators (`Spirit.Karma`) only. As a rule of thumb: it has to be implemented whenever a certain iterator type belongs to a container which is to be passed as an attribute to a generator normally exposing a STL container, and if the container type does not expose the interface of a STL container (i.e. `is_container<C>::type` would normally return `mpl::false_`).

### Related Attribute Customization Points

If this customization point is implemented, the following other customization points might need to be implemented as well.

<table>
<thead>
<tr>
<th>Name</th>
<th>When to implement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>traits::is_container</code></td>
<td>Needs to be implemented whenever a type is to be used as a container attribute in <code>Karma</code>.</td>
</tr>
<tr>
<td><code>traits::container_iterator</code></td>
<td><code>Karma: List (\%), Kleene (unary *), Plus (unary +), Repeat.</code></td>
</tr>
<tr>
<td><code>traits::begin_container</code></td>
<td><code>Karma: List (\%), Kleene (unary *), Plus (unary +), Repeat.</code></td>
</tr>
<tr>
<td><code>traits::end_container</code></td>
<td><code>Karma: List (\%), Kleene (unary *), Plus (unary +), Repeat.</code></td>
</tr>
<tr>
<td><code>traits::derefer_iterator</code></td>
<td><code>Karma: List (\%), Kleene (unary *), Plus (unary +), Repeat.</code></td>
</tr>
<tr>
<td><code>traits::next_iterator</code></td>
<td><code>Karma: List (\%), Kleene (unary *), Plus (unary +), Repeat.</code></td>
</tr>
<tr>
<td><code>traits::compare_iterators</code></td>
<td><code>Karma: List (\%), Kleene (unary *), Plus (unary +), Repeat.</code></td>
</tr>
</tbody>
</table>

### Example

Here are the header files needed to make the example code below compile:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <iostream>
#include <string>
#include <vector>
```

The example (for the full source code please see here: `customize_use_as_container.cpp`) uses the data structure
namespace client
{
    struct use_as_container
    {
        // Expose a pair holding a pointer to the use_as_container and to the
        // current element as our iterator.
        // We intentionally leave out having it a 'operator==()' to demonstrate
        // the use of the 'compare_iterators' customization point.
        struct iterator
        {
            iterator(use_as_container const* container, int const* current)
                : container_(container), current_(current)
            {}

            use_as_container const* container_;
            int const* current_;
        };

        // expose 'int' as the type of each generated element
        typedef int type;

        use_as_container(int value1, int value2, int value3)
            : value1_(value1), value2_(value2), value3_(value3)
        {}

        int value1_;
        std::string dummy1_; // insert some unrelated data
        int value2_;
        std::string dummy2_; // insert some more unrelated data
        int value3_;
    };
}

as a direct attribute to the List ($) generator. This type does not expose any of the interfaces of an STL container. It does not even expose the usual semantics of a container. The purpose of this artificial example is to demonstrate how the customization points can be used to expose independent data elements as a single container. The example shows how to enable its use as an attribute to Karma's repetitive generators.

In order to make this data structure compatible we need to specialize a couple of attribute customization points: traits::is_container, traits::container_iterator, traits::begin_container, and traits::end_container. In addition, we specialize all of the iterator related customization points as well: traits::deref_iterator, traits::next_iterator, and traits::compare_iterators.
// All specializations of attribute customization points have to be placed into
// the namespace boost::spirit::traits.
//
// Note that all templates below are specialized using the 'const' type.
// This is necessary as all attributes in Karma are 'const'.
namespace boost { namespace spirit { namespace traits
{
    // The specialization of the template 'is_container<>' will tell the
    // library to treat the type 'client::use_as_container' as a
    // container holding the items to generate output from.
    template <>
    struct is_container<client::use_as_container const>
    : mpl::true_
    {};

    // The specialization of the template 'container_iterator<>' will be
    // invoked by the library to evaluate the iterator type to be used
    // for iterating the data elements in the container. We simply return
    // the type of the iterator exposed by the embedded 'std::vector<int>'.
    template <>
    struct container_iterator<client::use_as_container const>
    {
        typedef client::use_as_container::iterator type;
    };

    // The specialization of the templates 'begin_container<>' and
    // 'end_container<>' below will be used by the library to get the iterators
    // pointing to the begin and the end of the data to generate output from.
    //
    // The passed argument refers to the attribute instance passed to the list
    // generator.
    template <>
    struct begin_container<client::use_as_container const>
    {
        static client::use_as_container::iterator
        call(client::use_as_container const& c)
        {
            return client::use_as_container::iterator(&c, &c.value1_);
        }
    };

    template <>
    struct end_container<client::use_as_container const>
    {
        static client::use_as_container::iterator
call(client::use_as_container const& c)
{
    return client::use_as_container::iterator(&c, (int const*)0);
};
}}

// All specializations of attribute customization points have to be placed into
// the namespace boost::spirit::traits.
namespace boost { namespace spirit { namespace traits
{
    // The specialization of the template 'deref_iterator<>' will be used to
dereference the iterator associated with our counter data structure.
template <>
struct deref_iterator<client::use_as_container::iterator>
{
    typedef client::use_as_container::type type;

    static type call(client::use_as_container::iterator const& it)
    {
        return *it.current_;
    }
};

template <>
struct next_iterator<client::use_as_container::iterator>
{
    static void call(client::use_as_container::iterator& it)
    {
        if (it.current_ == &it.container_->value1_)
            it.current_ = &it.container_->value2_;
        else if (it.current_ == &it.container_->value2_)
            it.current_ = &it.container_->value3_;
        else
            it.current_ = 0;
    }
};

template <>
struct compare_iterators<client::use_as_container::iterator>
{
    static bool call(client::use_as_container::iterator const& it1,
        client::use_as_container::iterator const& it2)
    {
        return it1.current_ == it2.current_ &&
            it1.container_ == it2.container_;
    }
};
}}

The last code snippet shows an example using an instance of the data structure client::use_as_container to generate output from a List ($) generator:

client::use_as_container d2 (1, 2, 3);
// use the instance of a 'client::use_as_container' instead of a STL vector
std::cout << karma::format(karma::int_ %", ", d2) << std::endl; // prints: '1, 2, 3'

As you can see, the specializations for the customization points as defined above enable the seamless integration of the custom data structure without having to modify the output format or the generator itself.
Dereference the Iterator pointing into a Container Attribute
deref_iterator

The template deref_iterator is a type used as an attribute customization point. It is invoked by the Karma repetitive generators (such as List (%), Kleene (unary *), Plus (unary +), and Repeat) in order to dereference an iterator pointing to an element of a container holding the attributes to generate output from.

Module Headers

```cpp
#include <boost/spirit/home/support/container.hpp>
```

Also, see Include Structure.

Note

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

Namespace

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::traits</td>
</tr>
</tbody>
</table>

Synopsis

```cpp
template <typename Iterator, typename Enable>
struct deref_iterator
{
    typedef <unspecified> type;
    static type call(Iterator& it);
};
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator</td>
<td>The type, Iterator of the iterator to dereference. This is the same as the type returned by the customization point traits::container_iterator.</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of deref_iterator utilizing SFINAE (i.e. boost::enable_if or boost::disable_if).</td>
<td>void</td>
</tr>
</tbody>
</table>

Notation

| Iterator | An iterator type. |
| it       | An instance of an iterator of type Iterator. |
A container type whose iterator type is `Iterator`.

**Expression Semantics**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>deref_iterator&lt;Iterator&gt;::type</td>
<td>Metafunction result evaluating to the type returned by dereferencing the iterator.</td>
</tr>
<tr>
<td>deref_iterator&lt;Iterator&gt;::call(it)</td>
<td>Return the element in the container referred to by the iterator. The type of the returned value is the same as returned by the metafunction result type.</td>
</tr>
</tbody>
</table>

**Predefined Specializations**

`Spirit` predefines specializations of this customization point for several types. The following table lists those types together with the types returned by the embedded typedef type:

<table>
<thead>
<tr>
<th>Template Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator</td>
<td>The metafunction result type evaluates to <code>boost::detail::iterator_traits&lt;Iterator&gt;::reference</code> and the function call() returns *it.</td>
</tr>
<tr>
<td>unused_type const*</td>
<td>The metafunction result type evaluates to unused_type and the function call() returns unused.</td>
</tr>
</tbody>
</table>

**When to implement**

The customization point `deref_iterator` needs to be implemented for a specific iterator type whenever the container this iterator belongs to is to be used as an attribute in place of a STL container. It is applicable for generators (`Spirit.Karma`) only. As a rule of thumb: it has to be implemented whenever a certain iterator type belongs to a container which is to be passed as an attribute to a generator normally exposing a STL container, `C` and if the container type does not expose the interface of a STL container (i.e. `is_container<C>::type` would normally return `mpl::false_`).

**Related Attribute Customization Points**

If this customization point is implemented, the following other customization points might need to be implemented as well.

<table>
<thead>
<tr>
<th>Name</th>
<th>When to implement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>traits::is_container</code></td>
<td>Needs to be implemented whenever a type is to be used as a container attribute in <code>Karma</code>.</td>
</tr>
<tr>
<td><code>traits::container_iterator</code></td>
<td>Karma: List (%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td><code>traits::begin_container</code></td>
<td>Karma: List (%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td><code>traits::end_container</code></td>
<td>Karma: List (%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td><code>traits::deref_iterator</code></td>
<td>Karma: List (%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td><code>traits::next_iterator</code></td>
<td>Karma: List (%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
<tr>
<td><code>traits::compare_iterators</code></td>
<td>Karma: List (%), Kleene (unary *), Plus (unary +), Repeat.</td>
</tr>
</tbody>
</table>
Example

Here are the header files needed to make the example code below compile:

```cpp
#include <boost/spirit/include/karma.hpp>
#include <iostream>
#include <vector>
```

The example (for the full source code please see here: customize_counter.cpp) uses the data structure

```cpp
namespace client
{
    struct counter
    {
        // expose the current value of the counter as our iterator
        typedef int iterator;

        // expose 'int' as the type of each generated element
        typedef int type;

        counter(int max_count)
            : counter_(0), max_count_(max_count)
        {}

        int counter_; // expose the current value of the counter as our iterator
        int max_count_; // expose 'int' as the type of each generated element
    }
}
```

as a direct attribute to the `List` generator. This type does not expose any of the interfaces of an STL container. It does not even expose the usual semantics of a container. The presented customization points build a counter instance which is incremented each time it is accessed. The examples shows how to enable its use as an attribute to `Karma's` repetitive generators.

In order to make this data structure compatible we need to specialize a couple of attribute customization points: `traits::is_container`, `traits::container_iterator`, `traits::begin_container`, and `traits::end_container`. In addition, we specialize one of the iterator related customization points as well: `traits::deref_iterator`. 


// All specializations of attribute customization points have to be placed into
// the namespace boost::spirit::traits.
//
// Note that all templates below are specialized using the 'const' type.
// This is necessary as all attributes in Karma are 'const'.
namespace boost { namespace spirit { namespace traits
{
    // The specialization of the template 'is_container<>' will tell the
    // library to treat the type 'client::counter' as a container providing
    // the items to generate output from.
    template <>
    struct is_container<client::counter const>
      : mpl::true_
{);

    // The specialization of the template 'container_iterator<>' will be
    // invoked by the library to evaluate the iterator type to be used
    // for iterating the data elements in the container.
    template <>
    struct container_iterator<client::counter const>
    {
        typedef client::counter::iterator type;
    };

    // The specialization of the templates 'begin_container<>' and
    // 'end_container<>' below will be used by the library to get the iterators
    // pointing to the begin and the end of the data to generate output from.
    // These specializations respectively return the initial and maximum
    // counter values.
    //
    // The passed argument refers to the attribute instance passed to the list
    // generator.
    template <>
    struct begin_container<client::counter const>
    {
        static client::counter::iterator
        call(client::counter const& c)
        {
            return c.counter_;  
        }
    };

    template <>
    struct end_container<client::counter const>
    {
        static client::counter::iterator
    }
```cpp
call(client::counter const& c)
{
    return c.max_count_;  // The specialization of the template 'deref_iterator<>' will be used to
    // dereference the iterator associated with our counter data structure.
    // Since we expose the current value as the iterator we just return the
    // current iterator as the return value.
}
```
Note
This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

Namespace

Name
boost::spirit::traits

Synopsis

```cpp
template <typename Iterator, typename Enable>
struct compare_iterators
{
    static bool call(Iterator const& it1, Iterator const& it2);
};
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator</td>
<td>The type, <code>Iterator</code> of the iterator to dereference. This is the same as the type returned by the customization point <code>traits::container_iterator</code>.</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of <code>compare_iterators</code> utilizing SFINAE (i.e. <code>boost::enable_if</code> or <code>boost::disable_if</code>).</td>
<td>void</td>
</tr>
</tbody>
</table>

Notation

- **Iterator**: An iterator type.
- **it1, it2**: Instances of an iterator of type `Iterator`.
- **C**: A container type whose iterator type is `Iterator`.

Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>compare_iterators&lt;Iterator&gt;::call(it1, it2)</code></td>
<td>Returns whether the iterators <code>it1 it2</code> are to be treated as being equal.</td>
</tr>
</tbody>
</table>

Predefined Specializations

Spirit predefines specializations of this customization point for several types. The following table lists those types together with the types returned by the embedded typedef `type`:
### Template Parameters

<table>
<thead>
<tr>
<th>Iterator</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The function call() returns it1 == it2.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unused_type const*</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The function call() always returns false.</td>
</tr>
</tbody>
</table>

### When to implement

The customization point `compare_iterators` needs to be implemented for a specific iterator type whenever the container this iterator belongs to is to be used as an attribute in place of a STL container. It is applicable for generators (`Spirit.Karma`) only. As a rule of thumb: it has to be implemented whenever a certain iterator type belongs to a container which is to be passed as an attribute to a generator normally exposing a STL container, C and if the container type does not expose the interface of a STL container (i.e. `is_container<C>::type` would normally return `mpl::false_`).

### Related Attribute Customization Points

If this customization point is implemented, the following other customization points might need to be implemented as well.

<table>
<thead>
<tr>
<th>Name</th>
<th>When to implement</th>
</tr>
</thead>
<tbody>
<tr>
<td>traits::is_container</td>
<td>Needs to be implemented whenever a type is to be used as a container attribute in <code>Karma</code>.</td>
</tr>
<tr>
<td>traits::container_iterator</td>
<td><code>Karma::List(%)</code>, <code>Kleene (unary *)</code>, <code>Plus (unary +)</code>, Repeat.</td>
</tr>
<tr>
<td>traits::begin_container</td>
<td><code>Karma::List(%)</code>, <code>Kleene (unary *)</code>, <code>Plus (unary +)</code>, Repeat.</td>
</tr>
<tr>
<td>traits::end_container</td>
<td><code>Karma::List(%)</code>, <code>Kleene (unary *)</code>, <code>Plus (unary +)</code>, Repeat.</td>
</tr>
<tr>
<td>traits::derefer_iterator</td>
<td><code>Karma::List(%)</code>, <code>Kleene (unary *)</code>, <code>Plus (unary +)</code>, Repeat.</td>
</tr>
<tr>
<td>traits::next_iterator</td>
<td><code>Karma::List(%)</code>, <code>Kleene (unary *)</code>, <code>Plus (unary +)</code>, Repeat.</td>
</tr>
<tr>
<td>traits::compare_iterators</td>
<td><code>Karma::List(%)</code>, <code>Kleene (unary *)</code>, <code>Plus (unary +)</code>, Repeat.</td>
</tr>
</tbody>
</table>

### Example

For an example of how to use the customization point `compare_iterators` please see here: `use_as_container`.

### Extract a C-Style String to Generate Output from a String Type (Karma)

### Determine if a Type is a Character

`is_char`

`is_char` is a metafunction that detects if a given type is a character.

### Module Headers

```cpp
#include <boost/spirit/home/support/string_traits.hpp>
```

Also, see Include Structure.
**Note**

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

**Namespace**

**Name**

boost::spirit::traits

**Synopsis**

```cpp
template <typename T>
struct is_char
{
    <unspecified>;
};
```

**Template parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>The type to detect.</td>
<td>none</td>
</tr>
</tbody>
</table>

**Notation**

T  An arbitrary type

**Expression Semantics**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_char&lt;T&gt;::type</td>
<td>mpl::true_ if T should be treated as a character type, and mpl::false_ otherwise. Generally, any implementation of is_char needs to behave as if if was a MPL Boolean Constant.</td>
</tr>
</tbody>
</table>

**Predefined Specializations**

<table>
<thead>
<tr>
<th>Type</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>mpl::false_</td>
</tr>
<tr>
<td>T const</td>
<td>is_char&lt;T&gt;</td>
</tr>
<tr>
<td>char</td>
<td>mpl::true_</td>
</tr>
<tr>
<td>wchar_t</td>
<td>mpl::true_</td>
</tr>
</tbody>
</table>

**When to implement**

This customization point needs to be implemented for any strings that use a type other than char or wchar_t to store character data.
Related Attribute Customization Points

If this customization point is implemented, the following other customization points need to be implemented as well.

<table>
<thead>
<tr>
<th>Name</th>
<th>When to implement</th>
</tr>
</thead>
<tbody>
<tr>
<td>traits::is_string</td>
<td>Whenever is_char is implemented.</td>
</tr>
<tr>
<td>traits::char_type_of</td>
<td>Whenever is_char is implemented.</td>
</tr>
</tbody>
</table>

Determine the Character Type of a String

char_type_of

This customization point is an MPL metafunction which returns the character type of a given string type. char_type_of handles user-defined types such as std::string, as well as C-style strings.

Module Headers

```cpp
#include <boost/spirit/home/support/string_traits.hpp>
```

Also, see Include Structure.

Note

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

Namespace

Name

boost::spirit::traits

Synopsis

```cpp
template <typename T>
struct char_type_of
{
    typedef <unspecified> type;
};
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>A string type.</td>
<td>none</td>
</tr>
</tbody>
</table>

Notation

T        An arbitrary type.
N        An arbitrary integral constant.
Char     A character type.
Traits A character traits type.

Allocator A standard allocator type.

**Expression Semantics**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>char_type_of&lt;T&gt;::type</td>
<td>The character type of the string type T.</td>
</tr>
</tbody>
</table>

**Predefined Specializations**

<table>
<thead>
<tr>
<th>Type</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>T const</td>
<td>Returns char_type_of&lt;T&gt;.</td>
</tr>
<tr>
<td>char</td>
<td>Returns char.</td>
</tr>
<tr>
<td>wchar_t</td>
<td>Returns wchar_t.</td>
</tr>
<tr>
<td>char const*</td>
<td>Returns char const.</td>
</tr>
<tr>
<td>wchar_t const*</td>
<td>Returns wchar_t const.</td>
</tr>
<tr>
<td>char*</td>
<td>Returns char.</td>
</tr>
<tr>
<td>wchar_t*</td>
<td>Returns wchar_t.</td>
</tr>
<tr>
<td>char[N]</td>
<td>Returns char.</td>
</tr>
<tr>
<td>wchar_t[N]</td>
<td>Returns wchar_t.</td>
</tr>
<tr>
<td>char const[N]</td>
<td>Returns char const.</td>
</tr>
<tr>
<td>wchar_t const[N]</td>
<td>Returns wchar_t const.</td>
</tr>
<tr>
<td>char(&amp;)[N]</td>
<td>Returns char.</td>
</tr>
<tr>
<td>wchar_t(&amp;)[N]</td>
<td>Returns wchar_t.</td>
</tr>
<tr>
<td>char const(&amp;)[N]</td>
<td>Returns char const.</td>
</tr>
<tr>
<td>wchar_t const(&amp;)[N]</td>
<td>Returns wchar_t const.</td>
</tr>
<tr>
<td>std::basic_string&lt;Char, Traits, Allocator&gt;</td>
<td>Returns Char.</td>
</tr>
</tbody>
</table>
### Name
<table>
<thead>
<tr>
<th>When to implement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>traits::is_char</strong></td>
</tr>
<tr>
<td><strong>traits::is_string</strong></td>
</tr>
<tr>
<td><strong>traits::extract_c_string</strong></td>
</tr>
</tbody>
</table>

### Get a C-style String from a String Type

**extract_c_string**

`extract_c_string` returns a pointer to an array of elements of a const character type. It is invoked through a static method `call`. This customization point is responsible for handling it's own garbage collecting; the lifetime of the returned C-string must be no shorter than the lifetime of the string instance passed to the `call` method.

### Module Headers

```
#include <boost/spirit/home/support/string_traits.hpp>
```

Also, see [Include Structure](#).

### Note

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

### Namespace

#### Name

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::spirit::traits</td>
</tr>
</tbody>
</table>

### Synopsis

```
template <typename String>
struct extract_c_string
{
    typedef <unspecified> char_type;

    static char_type const* call (String const&);
};
```

### Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>A string type.</td>
<td>none</td>
</tr>
</tbody>
</table>

### Notation

- **T** An arbitrary type.
- **Char** A character type.
Traits  A character traits type.
Allocator  A standard allocator type.
str  A string instance.

Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract_c_string&lt;T&gt;::char_type</td>
<td>The return type of call.</td>
</tr>
<tr>
<td>extract_c_string&lt;T&gt;::call(str)</td>
<td>Extract a c-string of type char_type from str.</td>
</tr>
</tbody>
</table>

Predefined Specializations

<table>
<thead>
<tr>
<th>Type</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>call takes a parameter of type T const*, and returns it without modification. An overload of call takes a parameter of type T* and casts it to T const*, returning the result. char_type is char_type_of&lt;T&gt;::type.</td>
</tr>
<tr>
<td>T const</td>
<td>call takes a parameter str of type T const and returns extract_c_string&lt;T&gt;::call(str). char_type is char_type_of&lt;T&gt;::type.</td>
</tr>
<tr>
<td>T&amp;</td>
<td>call takes a parameter str of type T&amp; and returns extract_c_string&lt;T&gt;::call(str). char_type is char_type_of&lt;T&gt;::type.</td>
</tr>
<tr>
<td>T const&amp;</td>
<td>call takes a parameter str of type T const&amp; and returns extract_c_string&lt;T&gt;::call(str). char_type is char_type_of&lt;T&gt;::type.</td>
</tr>
<tr>
<td>std::basic_string&lt;Char, Traits, Allocator&gt;</td>
<td>call takes a parameter str and returns str.c_str(). char_type is Char.</td>
</tr>
</tbody>
</table>

When to implement

This customization point needs to be implemented whenever traits::is_string is implemented.

Related Attribute Customization Points

If this customization point is implemented, the following other customization points need to be implemented as well.

<table>
<thead>
<tr>
<th>Name</th>
<th>When to implement</th>
</tr>
</thead>
<tbody>
<tr>
<td>traits::is_char</td>
<td>For string types whose underlying character type is not char or wchar_t, is_char must be implemented.</td>
</tr>
<tr>
<td>traits::is_string</td>
<td>Whenever extract_c_string is implemented.</td>
</tr>
<tr>
<td>traits::char_type_of</td>
<td>Whenever extract_c_string is implemented.</td>
</tr>
</tbody>
</table>
Atomically Extract an Attribute Value from a Container (Karma)

**attribute_as**

attribute_as atomically extracts an instance of a type from another type. This customization point is used by the `as<T>[]` directive.

**Module Headers**

```cpp
#include <boost/spirit/home/support/attributes_fwd.hpp>
```

Also, see Include Structure.

---

**Note**

This header file does not need to be included directly by any user program as it is normally included by other Spirit header files relying on its content.

**Namespace**

**Name**

boost::spirit::traits

**Synopsis**

```cpp
template <typename T, typename Attribute, typename Enable = void>
struct attribute_as;
```

**Template parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>The type of the attribute natively exposed by the component the attribute_as is invoked from.</td>
<td>none</td>
</tr>
<tr>
<td>Attribute</td>
<td>The type of the attribute to be used to generate output from.</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of attribute_as utilizing SFINAE (i.e. boost::enable_if or boost::disable_if).</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notation**

attr An instance of type Attr. 
Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute_as&lt;T, Attribute&gt;::type</td>
<td>The result type of the extraction.</td>
</tr>
<tr>
<td>attribute_as&lt;T, Attribute&gt;::call(attr)</td>
<td>Extract and return an instance of type.</td>
</tr>
<tr>
<td>attribute_as&lt;T, Attribute&gt;::valid_as(attr)</td>
<td>Determine, at runtime, if the extraction of an instance of type from attr would cause an error.</td>
</tr>
</tbody>
</table>

Predefined Specializations

Spirit predefines specializations of this customization point for several types. The following table lists those types together with the types exposed and the corresponding semantics:

<table>
<thead>
<tr>
<th>Template Parameters</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused_type</td>
<td>The exposed typedef type is defined to unused_type. The function call() returns an instance of unused_type.</td>
</tr>
</tbody>
</table>

When to implement

This customization point may need to be implemented when using the as<T>[] directive.

Create Components from Attributes (Qi and Karma)

Spirit supports the creation of a default parser or a default generator from a given attribute type. It implements a minimal set of predefined mappings from different attribute types to parsers and generators (for a description of the predefined mappings see Additional Attribute Requirements for Parsers and Additional Attribute Requirements for Generators). The customization points described in this section (traits::create_parser and traits::create_generator) can be specialized to define additional mappings for custom data types.

Define a Custom Attribute Mapping for a Parser

create_parser

The template create_parser is a type used as an customization point. It is invoked by the Qi create_parser API function in order to create a custom mapping of the given data type to a parser expression. This parser expression will be returned from create_parser whenever the given data type is encountered.

Module Headers

```cpp
// forwards to <boost/spirit/home/qi/auto.hpp>
#include <boost/spirit/include/qi_auto.hpp>
```

Also, see Include Structure.

Namespace

Name

boost::spirit::traits
Synopsis

```cpp
template <typename T, typename Enable>
struct create_parser
{
    typedef <unspecified> type;
    static type const& call();
};
```

Template parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>The type, ( T ) for which a custom mapping to a parser should be established.</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of <code>create_generator</code> utilizing SFINAE (i.e. <code>boost::enable_if</code> or <code>boost::disable_if</code>).</td>
<td>void</td>
</tr>
</tbody>
</table>

Notation

- \( T \)  An arbitrary type.

Expression Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create_parser&lt;T&gt;::type</code></td>
<td>Defines the type of the parser expression returned from <code>call</code>.</td>
</tr>
<tr>
<td><code>create_parser&lt;T&gt;::call()</code></td>
<td>Returns a parser expression (usually this is a <code>proto::expression</code>) to be used as the default parser for the given type, ( T ).</td>
</tr>
</tbody>
</table>

Predefined Specializations

Spirit predefines specializations of this customization point for several types. All predefined mappings are listed here: Additional Attribute Requirements for Parsers.

Note

It is possible to overload the predefined mappings for the listed types by providing your own specialization of the `create_parser` customization point for the type to modify.

When to implement

The customization point `create_parser` needs to be implemented for a specific type whenever this type should be usable with the API function `create_parser` (which includes using the `qi::auto_` parser and the special API functions based on the automatic creation of the matching parser type).

Example

For an example of how to use the customization point `create_parser` please see here: Example for Using the `qi::auto_` Parser.
Define a Custom Attribute Mapping for a Generator

**create_generator**

The template `create_generator` is a type used as a customization point. It is invoked by the `Karma create_generator` API function in order to create a custom mapping of the given data type to a generator expression. This generator expression will be returned from `create_generator` whenever the given data type is encountered.

**Module Headers**

```cpp
// forwards to <boost/spirit/home/karma/auto.hpp>
#include <boost/spirit/include/karma_auto.hpp>
```

Also, see Include Structure.

**Namespace**

**Name**

`boost::spirit::traits`

**Synopsis**

```
template <typename T, typename Enable>
struct create_generator
{
    typedef <unspecified> type;
    static type const& call();
};
```

**Template parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>The type, T for which a custom mapping to a generator should be established.</td>
<td>none</td>
</tr>
<tr>
<td>Enable</td>
<td>Helper template parameter usable to selectively enable or disable certain specializations of <code>create_generator</code> utilizing SFINAE (i.e. <code>boost::enable_if</code> or <code>boost::disable_if</code>).</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notation**

T An arbitrary type.

**Expression Semantics**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create_generator&lt;T&gt;::type</code></td>
<td>Defines the type of the generator expression returned from <code>call</code>.</td>
</tr>
<tr>
<td><code>create_generator&lt;T&gt;::call()</code></td>
<td>Returns a generator expression (usually this is a <code>proto::expression</code>) to be used as the default generator for the given type, T.</td>
</tr>
</tbody>
</table>
Predefined Specializations

Spirit predefines specializations of this customization point for several types. All predefined mappings are listed here: Additional Attribute Requirements for Generators.

Note

It is possible to overload the predefined mappings for the listed types by providing your own specialization of the `create_generator` customization point for the type to modify.

When to implement

The customization point `create_generator` needs to be implemented for a specific type whenever this type should be usable with the API function `create_generator` (which includes using the `karma::auto_generator` and the special API functions based on the automatic creation of the matching generator type).

Example

For an example of how to use the customization point `create_generator` please see here: Example for Using the `karma::auto_generator`. 
Supporting Libraries

The multi pass iterator

Backtracking in *Spirit.Qi* requires the use of the following types of iterator: forward, bidirectional, or random access. Because of backtracking, input iterators cannot be used. Therefore, the standard library classes `std::istreambuf_iterator` and `std::istream_iterator`, that fall under the category of input iterators, cannot be used. Another input iterator that is of interest is one that wraps a lexer, such as LEX.

**Note**

In general, *Spirit.Qi* generates recursive descent parser which require backtracking parsers by design. For this reason we need to provide at least forward iterators to any of *Spirit.Qi*’s API functions. This is not an absolute requirement though. In the future, we shall see more deterministic parsers that require no more than 1 character (token) of lookahead. Such parsers allow us to use input iterators such as the `std::istream_iterator` as is.

Backtracking can be implemented only if we are allowed to save an iterator position, i.e. making a copy of the current iterator. Unfortunately, with an input iterator, there is no way to do so, and thus input iterators will not work with backtracking in *Spirit.Qi*. One solution to this problem is to simply load all the data to be parsed into a container, such as a vector or deque, and then pass the begin and end of the container to *Spirit.Qi*. This method can be too memory intensive for certain applications, which is why the `multi_pass` iterator was created.

**Using the multi_pass**

The `multi_pass` iterator will convert any input iterator into a forward iterator suitable for use with *Spirit.Qi*. `multi_pass` will buffer data when needed and will discard the buffer when its contents is not needed anymore. This happens either if only one copy of the iterator exists or if no backtracking can occur.

A grammar must be designed with care if the `multi_pass` iterator is used. Any rule that may need to backtrack, such as one that contains an alternative, will cause data to be buffered. The rules that are optimal to use are repetition constructs (as kleene and plus).

Sequences of the form `a >> b` will buffer data as well. This is different from the behavior of *Spirit.Classic* but for a good reason. Sequences need to reset the current iterator to its initial state if one of the components of a sequence fails to match. To compensate for this behavior we added functionality to the `expect` parsers (i.e. constructs like `a > b`). Expectation points introduce deterministic points into the grammar ensuring no backtracking can occur if they match. For this reason we clear the buffers of any `multi_pass` iterator on each expectation point, ensuring minimal buffer content even for large grammars.

**Important**

If you use an error handler in conjunction with the `expect` parser while utilizing a `multi_pass` iterator and you intend to use the error handler to force a `retry` or a `fail` (see the description of error handlers - FIXME: insert link), then you need to instantiate the error handler using `retry` or `fail`, for instance:

```cpp
rule r<iterator_type> r;
on_error<retry>(r, std::cout << phoenix::val("Error!"));
```

If you fail to do so the resulting code will trigger an assert statement at runtime.

Any rule that repeats, such as `kleene_star (*a)` or positive such as `(a)+`, will only buffer the data for the current repetition.

In typical grammars, ambiguity and therefore lookahead is often localized. In fact, many well designed languages are fully deterministic and require no lookahead at all. Peeking at the first character from the input will immediately determine the alternative branch to take. Yet, even with highly ambiguous grammars, alternatives are often of the form `*(a | b | c | d)`. The input iterator moves on and is never stuck at the beginning. Let's look at a Pascal snippet for example:
program =
    programHeading >> block >> '.
    ;
block =
    *( labelDeclarationPart
       | constantDefinitionPart
       | typeDefinitionPart
       | variableDeclarationPart
       | procedureAndFunctionDeclarationPart
    )
    >> statementPart
    ;

Notice the alternatives inside the Kleene star in the rule `block`. The rule gobbles the input in a linear manner and throws away the past history with each iteration. As this is fully deterministic LL(1) grammar, each failed alternative only has to peek 1 character (token). The alternative that consumes more than 1 character (token) is definitely a winner. After which, the Kleene star moves on to the next.

Now, after the lecture on the features to be careful with when using `multi_pass`, you may think that `multi_pass` is way too restrictive to use. That's not the case. If your grammar is deterministic, you can make use of the `flush_multi_pass` pseudo parser in your grammar to ensure that data is not buffered when unnecessary (`flush_multi_pass` is available from the Spirit.Qi parser Repository).

Here we present a minimal example showing a minimal use case. The `multi_pass` iterator is highly configurable, but the default policies have been chosen so that its easily usable with input iterators such as `std::istreambuf_iterator`. For the complete source code of this example please refer to `multi_pass.cpp`.
int main()
{
    namespace spirit = boost::spirit;
    using spirit::ascii::space;
    using spirit::ascii::char_;
    using spirit::qi::double_;
    using spirit::qi::eol;

    std::ifstream in("multi_pass.txt"); // we get our input from this file
    if (!in.is_open()) {
        std::cout << "Could not open input file: 'multi_pass.txt'" << std::endl;
        return -1;
    }

    typedef std::istreambuf_iterator<char> base_iterator_type;
    spirit::multi_pass<base_iterator_type> first =
        spirit::make_default_multi_pass(base_iterator_type(in));

    std::vector<double> v;
    bool result = spirit::qi::phrase_parse(first
        , spirit::make_default_multi_pass(base_iterator_type())
        , double_ >> '(' >> double_ ) // recognize list of doubles
        , space | '#' >> *(char_ - eol) >> eol // comment skipper
        , v); // data read from file

    if (!result) {
        std::cout << "Failed parsing input file!" << std::endl;
        return -2;
    }

    std::cout << "Successfully parsed input file!" << std::endl;
    return 0;
}

Using the flush_multi_pass parser

The Spirit Repository contains the flush_multi_pass parser component. This is usable in conjunction with the multi_pass iterator to minimize the buffering. It allows to insert explicit synchronization points into your grammar where it is safe to clear any stored input as it is ensured that no backtracking can occur at this point anymore.

When the flush_multi_pass parser is used with multi_pass, it will call multi_pass::clear_queue(). This will cause any buffered data to be erased. This also will invalidate all other copies of multi_pass and they should not be used. If they are, an boost::illegal_backtracking exception will be thrown.

The multi_pass Policies

The multi_pass iterator is a templated class configurable using policies. The description of multi_pass above is how it was originally implemented (before it used policies), and is the default configuration now. But, multi_pass is capable of much more. Because of the open-ended nature of policies, you can write your own policy to make multi_pass behave in a way that we never before imagined.

The multi_pass class has two template parameters:

The multi_pass template parameters

<table>
<thead>
<tr>
<th>Input</th>
<th>The type multi_pass uses to acquire it's input. This is typically an input iterator, or functor.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Policies</td>
<td>The combined policies to use to create an instance of a multi_pass iterator. This combined policy type is described below</td>
</tr>
</tbody>
</table>
It is possible to implement all of the required functionality of the combined policy in a single class. But it has shown to be more convenient to split this into four different groups of functions, i.e. four separate, but well coordinated policies. For this reason the multi_pass library implements a template `iterator_policies::default_policy` allowing to combine several different policies, each implementing one of the functionality groups:

**Table 12. Policies needed for default_policy template**

<table>
<thead>
<tr>
<th>Template Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OwnershipPolicy</td>
<td>This policy determines how multi_pass deals with it's shared components.</td>
</tr>
<tr>
<td>CheckingPolicy</td>
<td>This policy determines how checking for invalid iterators is done.</td>
</tr>
<tr>
<td>InputPolicy</td>
<td>A class that defines how multi_pass acquires its input. The InputPolicy is parameterized by the Input template parameter to the multi_pass.</td>
</tr>
<tr>
<td>StoragePolicy</td>
<td>The buffering scheme used by multi_pass is determined and managed by the StoragePolicy.</td>
</tr>
</tbody>
</table>

The multi_pass library contains several predefined policy implementations for each of the policy types as described above. First we will describe those predefined types. Afterwards we will give some guidelines how you can write your own policy implementations.

**Predefined policies**

All predefined multi_pass policies are defined in the namespace `boost::spirit::iterator_policies`. 
### Table 13. Predefined policy classes

<table>
<thead>
<tr>
<th>Class name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>InputPolicy classes</strong></td>
<td></td>
</tr>
<tr>
<td>input_iterator</td>
<td>This policy directs <code>multi_pass</code> to read from an input iterator of type <code>Input</code>.</td>
</tr>
<tr>
<td>buffering_input_iterator</td>
<td>This policy directs <code>multi_pass</code> to read from an input iterator of type <code>Input</code>. Additionally it buffers the last character received from the underlying iterator. This allows to wrap iterators not buffering the last character on their own (as <code>std::istreambuf_iterator</code>).</td>
</tr>
<tr>
<td>istream</td>
<td>This policy directs <code>multi_pass</code> to read from an input stream of type <code>Input</code> (usually a <code>std::basic_istream</code>).</td>
</tr>
<tr>
<td>lex_input</td>
<td>This policy obtains its input by calling <code>yylex()</code>, which would typically be provided by a scanner generated by Flex. If you use this policy your code must link against a Flex generated scanner.</td>
</tr>
<tr>
<td>functor_input</td>
<td>This input policy obtains its data by calling a functor of type <code>Input</code>. The functor must meet certain requirements. It must have a typedef called <code>result_type</code> which should be the type returned from <code>operator()</code>. Also, since an input policy needs a way to determine when the end of input has been reached, the functor must contain a static variable named <code>eof</code> which is comparable to a variable of type <code>result_type</code>.</td>
</tr>
<tr>
<td>split_functor_input</td>
<td>This is essentially the same as the <code>functor_input</code> policy except that the (user supplied) function object exposes separate unique and shared sub classes, allowing to integrate the functors' <code>unique</code> data members with the <code>multi_pass</code> data items held by each instance and its <code>shared</code> data members will be integrated with the <code>multi_pass</code> members shared by all copies.</td>
</tr>
<tr>
<td><strong>OwnershipPolicy classes</strong></td>
<td></td>
</tr>
<tr>
<td>ref_counted</td>
<td>This class uses a reference counting scheme. The <code>multi_pass</code> will delete its shared components when the count reaches zero.</td>
</tr>
<tr>
<td>first_owner</td>
<td>When this policy is used, the first <code>multi_pass</code> created will be the one that deletes the shared data. Each copy will not take ownership of the shared data. This works well for Spirit, since no dynamic allocation of iterators is done. All copies are made on the stack, so the original iterator has the longest lifespan.</td>
</tr>
<tr>
<td><strong>CheckingPolicy classes</strong></td>
<td></td>
</tr>
<tr>
<td>no_check</td>
<td>This policy does no checking at all.</td>
</tr>
</tbody>
</table>
This policy keeps around a buffer id, or a buffer age. Every time `clear_queue()` is called on a `multi_pass` iterator, it is possible that all other iterators become invalid. When `clear_queue()` is called, `buf_id_check` increments the buffer id. When an iterator is dereferenced, this policy checks that the buffer id of the iterator matches the shared buffer id. This policy is most effective when used together with the `split_std_deque` StoragePolicy. It should not be used with the `fixed_size_queue` StoragePolicy, because it will not detect iterator dereferences that are out of range.

This policy has not been implemented yet. When it is, it will keep track of all iterators and make sure that they are all valid. This will be mostly useful for debugging purposes as it will incur significant overhead.

Despite its name this policy keeps all buffered data in a `std::vector`. All data is stored as long as there is more than one iterator. Once the iterator count goes down to one, and the queue is no longer needed, it is cleared, freeing up memory. The queue can also be forcibly cleared by calling `multi_pass::clear_queue()`.

This policy keeps a circular buffer that is size \( N+1 \) and stores \( N \) elements. `fixed_size_queue` is a template with a `std::size_t` parameter that specified the queue size. It is your responsibility to ensure that \( N \) is big enough for your parser. Whenever the foremost iterator is incremented, the last character of the buffer is automatically erased. Currently there is no way to tell if an iterator is trailing too far behind and has become invalid. No dynamic allocation is done by this policy during normal iterator operation, only on initial construction. The memory usage of this StoragePolicy is set at \( N+1 \) bytes, unlike `split_std_deque`, which is unbounded.

### Combinations: How to specify your own custom `multi_pass`

The beauty of policy based designs is that you can mix and match policies to create your own custom iterator by selecting the policies you want. Here's an example of how to specify a custom `multi_pass` that wraps an `std::istream_iterator<char>`, and is slightly more efficient than the default `multi_pass` (as generated by the `make_default_multi_pass()` API function) because it uses the `iterator_policies::first_owner OwnershipPolicy` and the `iterator_policies::no_check CheckingPolicy`:

```cpp
typedef multi_pass<
    std::istream_iterator<char>,
    iterator_policies::default_policy<
        iterator_policies::first_owner,
        iterator_policies::no_check,
        iterator_policies::buffering_input_iterator,
        iterator_policies::split_std_deque
    >
> first_owner_multi_pass_type;
```

The default template parameters for `iterator_policies::default_policy` are:
• iterator_policies::ref_counted OwnershipPolicy

• iterator_policies::no_check CheckingPolicy, if BOOST_SPIRIT_DEBUG is defined: iterator_policies::buf_id_check CheckingPolicy

• iterator_policies::buffering_input_iterator InputPolicy, and

• iterator_policies::split_std_deque StoragePolicy.

So if you use multi_pass<std::istream_iterator<char>> you will get those pre-defined behaviors while wrapping an std::istream_iterator<char>.

### Dealing with constant look ahead

There is one other pre-defined class called look_ahead. The class look_ahead is another predefine multi_pass iterator type. It has two template parameters: Input, the type of the input iterator to wrap, and a std::size_t N, which specifies the size of the buffer to the fixed_size_queue policy. While the default multi_pass configuration is designed for safety, look_ahead is designed for speed. look_ahead is derived from a multi_pass with the following policies: input_iterator InputPolicy, first_owner OwnershipPolicy, no_check CheckingPolicy, and fixed_size_queue<N> StoragePolicy.

This iterator is defined by including the files:

```cpp
// forwards to <boost/spirit/home/support/look_ahead.hpp>
#include <boost/spirit/include/support_look_ahead.hpp>
```

Also, see Include Structure.

### Reading from standard input streams

Yet another predefined iterator for wrapping standard input streams (usually a std::basic_istream<>) is called basic_istream_iterator<Char, Traits>. This class is usable as a drop in replacement for std::istream_iterator<Char, Traits>. Its only difference is that it is a forward iterator (instead of the std::istream_iterator, which is an input iterator). basic_istream_iterator is derived from a multi_pass with the following policies: istream InputPolicy, ref_counted OwnershipPolicy, no_check CheckingPolicy, and split_std_deque StoragePolicy.

There exists an additional predefined typedef:

```cpp
typedef basic_istream_iterator<char, std::char_traits<char>> > istream_iterator;
```

This iterator is defined by including the files:

```cpp
// forwards to <boost/spirit/home/support/istream_iterator.hpp>
#include <boost/spirit/include/support_istream_iterator.hpp>
```

Also, see Include Structure.

#### How to write a functor for use with the functor_input InputPolicy

If you want to use the functor_input InputPolicy, you can write your own function object that will supply the input to multi_pass. The function object must satisfy several requirements. It must have a typedef result_type which specifies the return type of its operator(). This is standard practice in the STL. Also, it must supply a static variable called eof which is compared against to know whether the input has reached the end. Last but not least the function object must be default constructible. Here is an example:
// define the function object
class iterate_a2m
{
public:
    typedef char result_type;

    iterate_a2m() : c('A') {}
    iterate_a2m(char c) : c_(c) {}

    result_type operator()() const
    {
        if (c_ == 'M')
            return eof;
        return c_++;
    }

    static result_type eof;

private:
    char c_;}

iterate_a2m::result_type iterate_a2m::eof = iterate_a2m::result_type('\0');

// create two iterators using the define function object, one of which is
// an end iterator
typedef multi_pass<iterate_a2m,
    iterator_policies::functor_input,
    iterator_policies::first_owner,
    iterator_policies::no_check,
    iterator_policies::split_std_deque>
functor_multi_pass_type;

functor_multi_pass_type first = functor_multi_pass_t(iterate_a2m());
functor_multi_pass_type last;

// use the iterators: this will print "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
while (first != last) {
    std::cout << *first;
    ++first;
}

How to write policies for use with multi_pass

All policies to be used with the default_policy template need to have two embedded classes: unique and shared. The unique class needs to implement all required functions for a particular policy type. In addition it may hold all member data items being unique for a particular instance of a multi_pass (hence the name). The shared class does not expose any member functions (except sometimes a constructor), but it may hold all member data items to be shared between all copies of a particular multi_pass.

InputPolicy

An InputPolicy must have the following interface:
```cpp
struct input_policy
{
    // Input is the same type used as the first template parameter
    // while instantiating the multi_pass
    template <typename Input>
    struct unique
    {
        // these typedef's will be exposed as the multi_pass iterator
        // properties
        typedef __unspecified_type__ value_type;
        typedef __unspecified_type__ difference_type;
        typedef __unspecified_type__ distance_type;
        typedef __unspecified_type__ pointer;
        typedef __unspecified_type__ reference;

        unique() {}
        explicit unique(Input) {}

        // destroy is called whenever the last copy of a multi_pass is
        // destructed (ownership_policy::release() returned true)
        //
        // mp:    is a reference to the whole multi_pass instance
        template <typename MultiPass>
        static void destroy(MultiPass& mp);

        // swap is called by multi_pass::swap()
        void swap(unique&);

        // get_input is called whenever the next input character/token
        // should be fetched.
        //
        // mp:    is a reference to the whole multi_pass instance
        //
        // This method is expected to return a reference to the next
        // character/token
        template <typename MultiPass>
        static typename MultiPass::reference get_input(MultiPass& mp);

        // advance_input is called whenever the underlying input stream
        // should be advanced so that the next call to get_input will be
        // able to return the next input character/token
        //
        // mp:    is a reference to the whole multi_pass instance
        template <typename MultiPass>
        static void advance_input(MultiPass& mp);

        // input_at_eof is called to test whether this instance is a
        // end of input iterator.
        //
        // mp:    is a reference to the whole multi_pass instance
        //
        // This method is expected to return true if the end of input is
        // reached. It is often used in the implementation of the function
        // storage_policy::is_eof.
        template <typename MultiPass>
        static bool input_at_eof(MultiPass const& mp);

        // input_is_valid is called to verify if the parameter t represents
        // a valid input character/token
        //
        // mp:    is a reference to the whole multi_pass instance
        //
        // t:     is the character/token to test for validity
        //
```
template <typename MultiPass>
static bool input_is_valid(MultiPass const& mp, value_type const& t);

// Input is the same type used as the first template parameter passed
// while instantiating the multi_pass
template <typename Input>
struct shared
{
  explicit shared(Input) {}
};

It is possible to derive the struct `unique` from the type `boost::spirit::detail::default_input_policy`. This type implements a minimal sufficient interface for some of the required functions, simplifying the task of writing a new input policy.

This class may implement a function `destroy()` being called during destruction of the last copy of a `multi_pass`. This function should be used to free any of the shared data items the policy might have allocated during construction of its `shared` part. Because of the way `multi_pass` is implemented any allocated data members in `shared` should not be deep copied in a copy constructor of `shared`.

**OwnershipPolicy**

The `OwnershipPolicy` must have the following interface:
It is possible to derive the struct `unique` from the type `boost::spirit::detail::default_ownership_policy`. This type implements a minimal sufficient interface for some of the required functions, simplifying the task of writing a new ownership policy.

This class may implement a function `destroy()` being called during destruction of the last copy of a `multi_pass`. This function should be used to free any of the shared data items the policy might have allocated during construction of its `shared` part. Because of the way `multi_pass` is implemented any allocated data members in `shared` should **not** be deep copied in a copy constructor of `shared`.

**CheckingPolicy**

The `CheckingPolicy` must have the following interface:
struct checking_policy
{
    struct unique
    {
        // swap is called by multi_pass::swap()
        void swap(unique&);
        
        // destroy is called whenever the last copy of a multi_pass is
        // destructed (ownership_policy::release() returned true)
        //
        // mp: is a reference to the whole multi_pass instance
        template <typename MultiPass>
        static void destroy(MultiPass & mp);
        
        // docheck is called before the multi_pass is dereferenced or
        // incremented.
        //
        // mp: is a reference to the whole multi_pass instance
        //
        // This method is expected to make sure the multi_pass instance is
        // still valid. If it is invalid an exception should be thrown.
        template <typename MultiPass>
        static void docheck(MultiPass const & mp);
        
        // clear_queue is called whenever the function
        // multi_pass::clear_queue is called on this instance
        //
        // mp: is a reference to the whole multi_pass instance
        template <typename MultiPass>
        static void clear_queue(MultiPass & mp);
    }
    
    struct shared {};
};

It is possible to derive the struct unique from the type boost::spirit::detail::default_checking_policy. This type implements a minimal sufficient interface for some of the required functions, simplifying the task of writing a new checking policy.

This class may implement a function destroy() being called during destruction of the last copy of a multi_pass. This function should be used to free any of the shared data items the policy might have allocated during construction of its shared part. Because of the way multi_pass is implemented any allocated data members in shared should not be deep copied in a copy constructor of shared.

StoragePolicy

A StoragePolicy must have the following interface:
struct storage_policy
{
    // Value is the same type as typename MultiPass::value_type
    template <typename Value>
    struct unique
    {
        // destroy is called whenever the last copy of a multi_pass is
        // destroyed (ownership_policy::release() returned true)
        // mp: is a reference to the whole multi_pass instance
        template <typename MultiPass>
        static void destroy(MultiPass& mp);

        // swap is called by multi_pass::swap()
        void swap(unique&);

        // dereference is called whenever multi_pass::operator*() is invoked
        // mp: is a reference to the whole multi_pass instance
        // This function is expected to return a reference to the current
        // character/token.
        template <typename MultiPass>
        static typename MultiPass::reference dereference(MultiPass const& mp);

        // increment is called whenever multi_pass::operator++ is invoked
        // mp: is a reference to the whole multi_pass instance
        template <typename MultiPass>
        static void increment(MultiPass& mp);

        // mp: is a reference to the whole multi_pass instance
        template <typename MultiPass>
        static void clear_queue(MultiPass& mp);

        // is_eof is called to test whether this instance is a end of input
        // iterator.
        // mp: is a reference to the whole multi_pass instance
        // This method is expected to return true if the end of input is
        // reached.
        template <typename MultiPass>
        static bool is_eof(MultiPass const& mp);

        // less_than is called whenever multi_pass::operator==() is invoked
        // mp: is a reference to the whole multi_pass instance
        // rhs: is the multi_pass reference this instance is compared
        // to
        // This function is expected to return true if the current instance
        // is equal to the right hand side multi_pass instance
        template <typename MultiPass>
        static bool equal_to(MultiPass const& mp, MultiPass const& rhs);

        // less_than is called whenever multi_pass::operator<() is invoked
        // mp: is a reference to the whole multi_pass instance
        // rhs: is the multi_pass reference this instance is compared
        // to
        //
// This function is expected to return true if the current instance
// is less than the right hand side multi_pass instance
template <typename MultiPass>
static bool less_than(MultiPass const& mp, MultiPass const& rhs);

// Value is the same type as typename MultiPass::value_type
template <typename Value>
struct shared {};

It is possible to derive the struct unique from the type boost::spirit::detail::default_storage_policy. This type implements a minimal sufficient interface for some of the required functions, simplifying the task of writing a new storage policy.

This class may implement a function destroy() being called during destruction of the last copy of a multi_pass. This function should be used to free any of the shared data items the policy might have allocated during construction of its shared part. Because of the way multi_pass is implemented any allocated data members in shared should not be deep copied in a copy constructor of shared.

Generally, a StoragePolicy is the trickiest policy to implement. You should study and understand the existing StoragePolicy classes before you try and write your own.

The line position iterator

The line_pos_iterator is a lightweight line position iterator. This iterator adapter only stores the current line number, nothing else. Unlike Spirit.Classic’s position_iterator, it does not store the column number and does not need an end iterator. The current column can be computed, if needed.

Class Reference

```cpp
template <class Iterator>
class line_pos_iterator : public boost::iterator_adaptor<
    line_pos_iterator<Iterator> // Derived,
    , Iterator // Base,
    , boost::use_default // Value,
    , boost::forward_traversal_tag // CategoryOrTraversal
> {
    public:
        line_pos_iterator();

        explicit line_pos_iterator(Iterator);

        std::size_t position() const;

    private:
        friend class boost::iterator_core_access;

        void increment();

        std::size_t line; // The line position.

        typename std::iterator_traits<Iterator>::value_type prev;
};
```

get_line

```cpp
template <class Iterator>
inline std::size_t get_line(Iterator);
```
Get the line position. Returns -1 if Iterator is not a line_pos_iterator.

**get_line_start**

```cpp
template <class Iterator>
inline Iterator get_line_start(Iterator lower_bound, Iterator current);
```

Get an iterator to the beginning of the line. Applicable to any iterator.

**get_current_line**

```cpp
template <class Iterator>
inline iterator_range<Iterator>
get_current_line(Iterator lower_bound, Iterator current, 
                 Iterator upper_bound);
```

Get an iterator_range containing the current line. Applicable to any iterator.

**get_column**

```cpp
template <class Iterator>
inline std::size_t get_column(Iterator lower_bound, Iterator current, 
                               std::size_t tabs = 4);
```

Get the current column. Applicable to any iterator.

## The utree data structure

*utree* is a dynamically-typed hierarchical data structure that can represent abstract syntax trees. It's well integrated with *Spirit.Qi* and *Spirit.Karma*. *utree* can be passed as an attribute to almost any grammar. *utree*'s type system is implemented through the use of a discriminated union and type punning.

*utree* has a minimal memory footprint. The data structure size is 16 bytes on a 32-bit platform, and 32 bytes on 64-bit a platform (4*sizeof(void*)). Being a container of itself, it can represent tree structures.

Each instance of an *utree* data structure can store exactly one of the following data types at a time:
struct utree_type
{
    enum info
    {
        invalid_type, // the utree has not been initialized (it's
                      // default constructed)
        nil_type, // nil is the sentinel (empty) utree type.
        list_type, // A doubly linked list of utrees.
        range_type, // A range of list::iterators.
        reference_type, // A reference to another utree.
        any_type, // A pointer or reference to any C++ type.
        function_type, // A utree holding a stored_function<F> object,
                        // where F is an unary function object taking a
                        // utree as it's parameter and returning a
                        // utree.

        // numeric atoms
        bool_type, // An utree holding a boolean value
        int_type, // An utree holding a integer (int) value
        double_type, // An utree holding a floating point (double) value

        // text atoms (utf8)
        string_type, // An UTF-8 string
        string_range_type, // A pair of iterators into an UTF-8 string
        symbol_type, // An UTF-8 symbol name
        binary_type // Arbitrary binary data
    };

typedef boost::uint_t<sizeof(info)*8>::exact exact_integral_type;
typedef boost::uint_t<sizeof(info)*8>::fast fast_integral_type;
};

The UTF-8 string, UTF-8 symbol, and binary data types are internally stored either directly as the node data (small string optimization applied), or they are allocated from the heap, storing the pointer to the allocated data in the utree. The maximum possible length of the data to be stored in the node data depends on the platform the utree is compiled for. It is 14 bytes for a 32-bit platform and 30 bytes for a 64-bit platform.

Class Reference

The utree data structure is very versatile and can be used as an attribute for all possible Spirit.Qi parsers and Spirit.Karma generators. For this reason, it exposes a set of typedef's making it compatible with STL containers:

typedef utree value_type;
typedef utree& reference;
typedef utree const& const_reference;
typedef std::ptrdiff_t difference_type;
typedef std::size_t size_type;
typedef detail::list::node_iterator<utree> iterator;
typedef detail::list::node_iterator<utree const> const_iterator;

The utree data type exposes the functional interface of a bidirectional STL container. The iterators returned from begin() et.al. conform to the Standard requirements of a bidirectional iterator.
// STL Container interface

// insertion
template <class T>
void push_back(T const&);
template <class T>
void push_front(T const&);
template <class T>
iterator insert(iterator, T const&);
template <class T>
void insert(iterator, std::size_t, T const&);
template <class Iterator>
void insert(iterator, Iterator, Iterator);

// erasure
void pop_front();
void pop_back();
iterator erase(iterator);
iterator erase(iterator, iterator);

// front access
reference front();
const_reference front() const;
iterator begin();
const_iterator begin() const;
ref_iterator ref_begin();

// back access
reference back();
const_reference back() const;
iterator end();
const_iterator end() const;
ref_iterator ref_end();

The exposed container interface makes the utree usable with all Spirit.Qi parser and Spirit.Karma generator components, which are compatible with an STL container attribute type.

A utree can be constructed or initialized from a wide range of data types, allowing to create utree instances for every possible node type (see the description of utree_type::info above). For this reason it exposes a constructor and an assignment operator for each of the allowed node types as shown below. All constructors are non-explicit on purpose, allowing to use an utree instance as the attribute to almost any Qi parser.
// This constructs an `invalid_type` node. When used in places
// where a boost::optional is expected (i.e. as an attribute for the
// optional component), this represents the 'empty' state.
utree(invalid_type = invalid_type());

// This initializes a `nil_type` node, which represents a valid,
// 'initialized empty' utree (different from invalid_type!).
ulree(nil_type);
reference operator=(nil_type);

// This initializes a `boolean_type` node, which can hold 'true' or
// 'false' only.
exlicit utree(bool);
reference operator=(bool);

// This initializes an `integer_type` node, which can hold arbitrary
// integers. For convenience these functions are overloaded for signed
// and unsigned integer types.
ulree(unsigned int);
ulree(int);
reference operator=(unsigned int);
reference operator=(int);

// This initializes a `double_type` node, which can hold arbitrary
// floating point (double) values.
ulree(double);
reference operator=(double);

// This initializes a `string_type` node, which can hold a narrow
// character sequence (usually an UTF-8 string).
ulree(char);
ulree(char const*);
ulree(char const*, std::size_t);
ulree(std::string const&);
reference operator=(char);
reference operator=(char const*);
reference operator=(std::string const&);

// This constructs a `string_range_type` node, which does not copy the
// data but stores the iterator range to the character sequence the
// range has been initialized from.
ulree(utf8_string_range_type const&,
shallow_tag);

// This initializes a `reference_type` node, which holds a reference to
// another utree node. All operations on such a node are automatically
// forwarded to the referenced utree instance.
ulree(boost::reference_wrapper<utree>);
reference operator=(boost::reference_wrapper<utree>);

// This initializes an `any_type` node, which can hold a pointer to an
// instance of any type together with the typeid of that type. When
// accessing that pointer the typeid will be checked, causing a
// std::bad_cast to be thrown if the typeids do not match.
ulree(any_ptr const&);
reference operator=(any_ptr const&);

// This initializes a `range_type` node, which holds an utree list node
// the elements of which are copy constructed (assigned) from the
// elements referenced by the given range of iterators.
template <class Iterator>
ulree(boost::iterator_range<Iterator>);
template <class Iterator>
reference operator=(boost::iterator_range<Iterator>);
This initializes a `function_type` node from a polymorphic function object pointer (takes ownership) or reference.

```
utree(function_base const&);
reference operator=(function_base const&);
utree(function_base*);
reference operator=(function_base*);
```

This initializes either a `string_type`, a `symbol_type`, or a `binary_type` node (depending on the template parameter `type_`), which will hold the corresponding narrow character sequence (usually an UTF-8 string).

```
template <class Base, utree_type::info type_>
utree(basic_string<Base, type_> const&);
template <class Base, utree_type::info type_>
reference operator=(basic_string<Base, type_> const&);
```

The `utree` data type exposes the functional interface compatible to `Boost.Variant` as well. Its very nature is to hold different data types, one at each point in time, making it functionally very similar to `Boost.Variant`.

```
// return the data type (`utree_type::info`) of the currently stored data item
utree_type::info which() const;

// access the currently stored data in a type safe manner, this will throw a `std::bad_cast()` if the currently stored data item is not default convertible to `T`.
template <class T>
T get() const;
```

The exposed variant-like interface makes the `utree` usable with all `Spirit.Qi` parser and `Spirit.Karma` generator components, which are compatible with an `Boost.Variant` attribute type.

### String Types

The `utree` string types described below are used by the `utree` API only. These are not used to store information in the `utree` itself. Their purpose is to refer to different internal `utree` node types only. For instance, creating a `utree` from a binary data type will create a `binary_type` `utree` node (see above).

The binary data type can be represented either verbatim as a sequence of bytes or as a pair of iterators into some other stored binary data sequence. Use this string type to access/create a `binary_type` `utree`.

```
typedef basic_string<boost::iterator_range<char const*>, utree_type::binary_type, binary_range_type>;
typedef basic_string<std::string, utree_type::binary_type, binary_string_type>;
```

The UTF-8 string can be represented either verbatim as a sequence of characters or as a pair of iterators into some other stored binary data sequence. Use this string type to access/create a `string_type` `utree`.

```
typedef basic_string<boost::iterator_range<char const*>, utree_type::string_type, utf8_string_range_type>;
typedef basic_string<std::string, utree_type::string_type, utf8_string_type>;
```
The UTF-8 symbol can be represented either verbatim as a sequence of characters or as a pair of iterators into some other stored binary data sequence. Use this string type to access/create a symbol_type utree.

```cpp
typedef basic_string<
    boost::iterator_range<char const*>, utree_type::symbol_type
> utf8_symbol_range_type;
typedef basic_string<
    std::string, utree_type::symbol_type
> utf8_symbol_type;
```

### Function Object Interface

The stored_function template class can to store a unary function objects with a signature of utree(scope const&) as a utree node.

```cpp
struct function_base
{
    virtual ~function_base() {}  
    virtual utree operator()(utree const& env) const = 0;
    virtual utree operator()(utree& env) const = 0;

    // Calling f.clone() must return a newly allocated function_base
    // instance that is equal to f.
    virtual function_base* clone() const = 0;
};

template <typename F>
struct stored_function : function_base
{
    F f;
    stored_function(F f = F());
    virtual ~stored_function();
    virtual utree operator()(utree const& env) const;
    virtual utree operator()(utree& env) const;
    virtual function_base* clone() const;
};

template <typename F>
struct referenced_function : function_base
{
    F& f;
    referenced_function(F& f);
    virtual ~referenced_function();
    virtual utree operator()(utree const& env) const;
    virtual utree operator()(utree& env) const;
    virtual function_base* clone() const;
};
```

### Exceptions

All exceptions thrown by utree are derived from utree_exception.

```cpp
struct utree_exception : std::exception {};
```

The bad_type_exception is thrown whenever somebody calls a member function, which applies to certain stored utree_type's only, but this precondition is violated as the utree instance holds some other type.

```cpp
struct bad_type_exception /*: utree_exception*/;
```
The `empty_exception` is thrown whenever a precondition of a list or range `utree` method is violated due to the list or range being empty.

```cpp
struct empty_exception /*: utree_exception*/;
```

**Example: Sexpr Parser**

Our first example demonstrates how to use `utree` to write a parser for symbolic expressions. While `utree` is capable of representing just about any AST, `utree`'s design is based on the simple yet powerful nature of symbolic expressions. This example introduces a number of basic and intermediate `utree` development techniques: using `Spirit.Qi` and `Spirit.Karma` integration, tracking source code locations and taking advantage of UTF8 support.

The source for this example can be found here: ../../example/support/utree.

```cpp
namespace sexpr
{

    template <typename Iterator, typename ErrorHandler = error_handler<Iterator> >
    struct parser : qi::grammar<Iterator, utree(), whitespace<Iterator> >
    {
        qi::rule<Iterator, utree(), whitespace<Iterator> >
            start, element, list;
        qi::rule<Iterator, utree>()
            atom;
        qi::rule<Iterator, int>()
            integer;
        qi::rule<Iterator, utf8_symbol_type>()
            symbol;
        qi::rule<Iterator, utree::nil_type>()
            nil_;
        qi::rule<Iterator, binary_string_type>()
            binary;
        utf8::parser<Iterator>
            string;
        px::function<ErrorHandler> const
            error;
        tagger<Iterator, save_line_pos>
            pos;

        parser(std::string const& source_file = "<string>"):
            parser::base_type(start), error(ErrorHandler(source_file))
        {  
            using standard::char_;  
            using qi::unused_type;  
            using qi::lexeme;  
            using qi::hex;  
            using qi::oct;  
            using qi::no_case;  
            using qi::real_parser;  
            using qi::strict_real_policies;  
            using qi::uint_parser;  
            using qi::bool_parser;  
            using qi::on_error;
```

506
using qi::fail;
using qi::int_;  
using qi::lit;
using qi::_val;
using qi::_1;
using qi::_2;
using qi::_3;
using qi::_4;

real_parser<double, strict_real_policies<double> > strict_double;
uint_parser<unsigned char, 16, 2, 2> hex2;
bool_parser<bool, sexpr::bool_input_policies> boolean;

start = element.alias();

element = atom | list;

list = pos(_val, '(') > *element > ');

atom = nil_
    | strict_double
    | integer
    | boolean
    | string
    | symbol
    | binary;

nil_ = qi::attr_cast(lit("nil"));

integer = lexeme[ no_case["#x"] > hex]
    | lexeme[ no_case["#o"] >> oct]
    | lexeme[-no_case["#d"] >> int_];

std::string exclude = std::string(" ()";"\x01-\x1f\x7f") + '\0';
symbol = lexeme[+(~char_(exclude))];

binary = lexeme["#" > 'hex2 > '#];

start.name("sexpr");
element.name("element");
list.name("list");
atom.name("atom");
nil_.name("nil");
integer.name("integer");
symbol.name("symbol");
binary.name("binary");

    on_error<fail>(start, error(_1, _2, _3, _4))

};

} // sexpr
**Spirit FAQ**

I'm getting multiple symbol definition errors while using Visual C++. Anything I could do about that?

Do you see strange multiple symbol definition linker errors mentioning `boost::mpl::failed` and `boost::spirit::qi::rule`? Then this FAQ entry might be for you.

Boost.Mpl implements a macro `BOOST_MPL_ASSERT_MSG()` which essentially is a more powerful version of static_assert. Unfortunately under certain circumstances using this macro may lead to the aforementioned linker errors.

Spirit allows you to define a preprocessor constant disabling the usage of `BOOST_MPL_ASSERT_MSG()`, while switching to `BOOST_STATIC_ASSERT()` instead. For that you need define `BOOST_SPIRIT_DONT_USE_MPL_ASSERT_MSG=1`. Do this by adding

```
-DBOOST_SPIRIT_DONT_USE_MPL_ASSERT_MSG=1
```

on the compiler command line or by inserting a

```
#define BOOST_SPIRIT_DONT_USE_MPL_ASSERT_MSG 1
```

into your code before any Spirit headers get included.

Using this trick has no adverse effects on any of the functionality of Spirit. The only change you might see while using this workaround is less verbose error messages generated from static_assert.

I'm very confused about the header hell in my boost/spirit directory. What's all this about?

The boost/spirit directory currently holds two versions of the Spirit library: Spirit.Classic (former V1.8.x) and SpiritV2. Both are completely independent and usually not used at the same time. Do not mix these two in the same grammar.

Spirit.Classic evolved over years in a fairly complex directory structure:

```
boost/spirit/actor
boost/spirit/attribute
boost/spirit/core
boost/spirit/debug
boost/spirit/dynamic
boost/spirit/error_handling
boost/spirit/iterator
boost/spirit/meta
boost/spirit/symbols
boost/spirit/tree
boost/spirit/utility
```

While introducing Spirit V2 we restructured the directory structure in order to accommodate two versions at the same time. All of Spirit.Classic now lives in the directory

```
boost/spirit/home/classic
```

where the directories above contain forwarding headers to the new location allowing to maintain application compatibility. The forwarding headers issue a warning (starting with Boost V1.38) telling the user to change their include paths. Please expect the above directories/forwarding headers to go away soon.
This explains the need for the directory

```
boost/spirit/include
```

which contains forwarding headers as well. But this time the headers won't go away. We encourage application writers to use only the includes contained in this directory. This allows us to restructure the directories underneath if needed without worrying application compatibility. Please use those files in your application only. If it turns out that some forwarding file is missing, please report this as a bug.

Spirit V2 is not about parsing only anymore (as *Spirit.Classic*). It now consists out of 3 parts (sub-libraries): *Spirit.Qi, Spirit.Karma,* and *Spirit.Lex.* The header files for those live in

```
boost/spirit/home/qi
boost/spirit/home/karma
boost/spirit/home/lex
```

and have forwarding headers in

```
boost/spirit/include
```

*Spirit.Qi* is the direct successor to *Spirit.Classic* as it implements a DSEL (domain specific embedded language) allowing to write parsers using the syntax of C++ itself (parsers in the sense turning a sequence of bytes into an internal data structure). It is not compatible with *Spirit.Classic,* the main concepts are similar, though.

*Spirit.Karma* is the counterpart to *Spirit.Qi.* It implements a similar DSEL but for writing generators (i.e. the things turning internal data structures into a sequence of bytes, most of the time - strings). *Spirit.Karma* is the Yang to *Spirit.Qi*’s Yin, it’s almost like a mirrored picture.

*Spirit.Lex* is (as the name implies) a library allowing to write lexical analyzers. These are either usable stand alone or can be used as a front end for *Spirit.Qi* parsers. If you know flex you shouldn’t have problems understanding *Spirit.Lex.* This library actually doesn't implement the lexer itself. All it does is to provide an interface to pre-existing lexical analyzers. Currently it’s using Ben Hansons excellent *Lexertl* library (proposed for a Boost review, BTW) as its underlying workhorse.

Again, don’t use any of the header files underneath the boost/spirit/home directory directly, always include files from the boost/spirit/include directory.

The last bit missing is *Boost.Phoenix* (which currently still lives under the Spirit umbrella, but already has been accepted as a Boost library, so it will move away). *Boost.Phoenix* is a library allowing to write functional style C++, which is interesting in itself, but as it initially has been developed for Spirit, it is nicely integrated and very useful when it comes to writing semantic actions. I think using the boost/spirit/include/phoenix... headers will be safe in the future as well, as we will probably redirect to the Boost.Phoenix headers as soon as these are available.

**Why doesn't my symbol table work in a no_case directive?**

In order to perform case-insensitive parsing (using *no_case*) with a symbol table (i.e. use a symbols<> parser in a no_case directive), that symbol table needs to be filled with all-lowercase contents. Entries containing one or more uppercase characters will not match any input.

**I'm getting a compilation error mentioning boost::function and/or boost::function4. What does this mean?**

If you are using Visual C++ and have an error like:
error C2664: 'bool boost::function4<R,T0,T1,T2,T3>::operator()(T0,T1,T2,T3) const' :
cannot convert parameter 4 from '...' to '...'

or you are using GCC and have an error like:

error: no match for call to '(const boost::function<bool ()(...)>) (...)'
note: candidates are: ...
boost::function4<R,T1,T2,T3,T4>::operator()(T0,T1,T2,T3) const [with ...]

then this FAQ entry may help you.

The definition of a Rule or Grammar may contain a skip parser type. If it does, it means that non-terminal can only be used with a skip parser of a compatible type. The error above arises when this is not the case, i.e.:

• a non-terminal defined with a skip parser type is used without a skip parser; for example, a rule with a skip parser type is used inside a `lexeme` directive, or a grammar with a skip parser type is used in `parse` instead of `phrase_parse`,

• or a non-terminal is used with a skip parser of an incompatible type; for example, a rule defined with one skip parser type calls a second rule defined with another, incompatible skip parser type.

**Note**

Notes

Porting from Spirit 1.8.x

The current version of Spirit is a complete rewrite of earlier versions (we refer to earlier versions as Spirit.Classic). The parser generators are now only one part of the whole library. The parser submodule of Spirit is now called Spirit.Qi. It is conceptually different and exposes a completely different interface. Generally, there is no easy (or automated) way of converting parsers written for Spirit.Classic to Spirit.Qi. Therefore this section can give only guidelines on how to approach porting your older parsers to the current version of Spirit.

Include Files

The overall directory structure of the Spirit directories is described in the section Include Structure and the FAQ entry Header Hell. This should give you a good overview on how to find the needed header files for your new parsers. Moreover, each section in the Qi Reference lists the required include files needed for any particular component.

It is possible to tell from the name of a header file, what version it belongs to. While all main include files for Spirit.Classic have the string 'classic_' in their name, for instance:

```cpp
#include <boost/spirit/include/classic_core.hpp>
```

we named all main include files for Spirit.Qi to have the string 'qi_' as part of their name, for instance:

```cpp
#include <boost/spirit/include/qi_core.hpp>
```

The following table gives a rough list of corresponding header file between Spirit.Classic and Spirit.Qi, but this can be used as a starting point only, as several components have either been moved to different submodules or might not exist in the newer version anymore. We list only include files for the topmost submodules. For header files required for more lower level components please refer to the corresponding reference documentation of this component.

<table>
<thead>
<tr>
<th>Include file in Spirit.Classic</th>
<th>Include file in Spirit.Qi</th>
</tr>
</thead>
<tbody>
<tr>
<td>classic.hpp</td>
<td>qi.hpp</td>
</tr>
<tr>
<td>classic_actor.hpp</td>
<td>none, use Boost.Phoenix for writing semantic actions</td>
</tr>
<tr>
<td>classic_attribute.hpp</td>
<td>none, use local variables for rules instead of closures, the primitives parsers now directly support lazy parameterization</td>
</tr>
<tr>
<td>classic_core.hpp</td>
<td>qi_core.hpp</td>
</tr>
<tr>
<td>classic_debug.hpp</td>
<td>qi_debug.hpp</td>
</tr>
<tr>
<td>classic_dynamic.hpp</td>
<td>none, use Spirit.Qi predicates instead of if_p, while_p, for_p (included by qi_core.hpp), the equivalent for lazy_p is now included by qi_auxiliary.hpp</td>
</tr>
<tr>
<td>classic_error_handling.hpp</td>
<td>none, included in qi_core.hpp</td>
</tr>
<tr>
<td>classic_meta.hpp</td>
<td>none</td>
</tr>
<tr>
<td>classic_symbols.hpp</td>
<td>none, included in qi_core.hpp</td>
</tr>
<tr>
<td>classic_utility.hpp</td>
<td>none, not part of Spirit.Qi anymore, these components will be added over time to the Repository</td>
</tr>
</tbody>
</table>
The Free Parse Functions

The free parse functions (i.e. the main parser API) has been changed. This includes the names of the free functions as well as their interface. In *Spirit.Classic* all free functions were named `parse`. In *Spirit.Qi* they are are named either `qi::parse` or `qi::phrase_parse` depending on whether the parsing should be done using a skipper (`qi::phrase_parse`) or not (`qi::parse`). All free functions now return a simple `bool`. A returned `true` means success (i.e. the parser has matched) or `false` (i.e. the parser didn't match). This is equivalent to the former old `parse_info` member `hit`. *Spirit.Qi* doesn't support tracking of the matched input length anymore. The old `parse_info` member `full` can be emulated by comparing the iterators after `qi::parse` returned.

All code examples in this section assume the following include statements and using directives to be inserted. For *Spirit.Classic*:

```cpp
#include <boost/spirit/include/classic.hpp>
#include <boost/spirit/include/phoenix1.hpp>
#include <iostream>
#include <string>
using namespace boost::spirit::classic;
```

and for *Spirit.Qi*:

```cpp
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <iostream>
#include <string>
#include <algorithm>
using namespace boost::spirit;
```

The following similar examples should clarify the differences. First the base example in *Spirit.Classic*:

```cpp
std::string input("1,1");
parse_info<std::string::iterator> pi = parse(input.begin(), input.end(), int_p);
if (pi.hit)
    std::cout << "successful match!\n";
if (pi.full)
    std::cout << "full match!\n";
else
    std::cout << "stopped at: " << std::string(pi.stop, input.end()) << "\n";
std::cout << "matched length: " << pi.length << "\n";
```

And here is the equivalent piece of code using *Spirit.Qi*:

```cpp
```
The changes required for phrase parsing (i.e. parsing using a skipper) are similar. Here is how phrase parsing works in *Spirit.Classic*:

```cpp
std::string input("1,1");
std::string::iterator it = input.begin();
bool result = qi::parse(it, input.end(), qi::int_);

if (result)
    std::cout << "successful match!\n";
if (it == input.end())
    std::cout << "full match!\n";
else
    std::cout << "stopped at: " << std::string(it, input.end()) << "\n";

// seldomly needed: use std::distance to calculate the length of the match
std::cout << "matched length: " << std::distance(input.begin(), it) << "\n";
```

And here the equivalent example in *Spirit.Qi*:

```cpp
std::string input(" 1, 1");
parse_info<std::string::iterator> pi = parse(input.begin(), input.end(), int_p, space_p);

if (pi.hit)
    std::cout << "successful match!\n";
if (pi.full)
    std::cout << "full match!\n";
else
    std::cout << "stopped at: " << std::string(pi.stop, input.end()) << "\n";
std::cout << "matched length: " << pi.length << "\n";
```

Note, how character parsers are in a separate namespace (here boost::spirit::ascii::space) as *Spirit.Qi* now supports working with different character sets. See the section Character Encoding Namespace for more information.

### Naming Conventions

In *Spirit.Classic* all parser primitives have suffixes appended to their names, encoding their type: "_p" for parsers, "_a" for lazy actions, "_d" for directives, etc. In *Spirit.Qi* we don’t have anything similar. The only suffixes are single underscore letters "_" applied where the name would otherwise conflict with a keyword or predefined name (such as int_ for the integer parser). Overall, most, if not all primitive parsers and directives have been renamed. Please see the Qi Quick Reference for an overview on the names for the different available parser primitives, directives and operators.
Parser Attributes

In *Spirit.Classic* most of the parser primitives don't expose a specific attribute type. Most parsers expose the pair of iterators pointing to the matched input sequence. As in *Spirit.Qi* all parsers expose a parser specific attribute type it introduces a special directive `raw[]` allowing to achieve a similar effect as in *Spirit.Classic*. The `raw[]` directive exposes the pair of iterators pointing to the matching sequence of its embedded parser. Even if we very much encourage you to rewrite your parsers to take advantage of the generated parser specific attributes, sometimes it is helpful to get access to the underlying matched input sequence.

Grammars and Rules

The `grammar<>` and `rule<>` types are of equal importance to *Spirit.Qi* as they are for *Spirit.Classic*. Their main purpose is still the same: they allow to define non-terminals and they are the main building blocks for more complex parsers. Nevertheless, both types have been redesigned and their interfaces have changed. Let's have a look at two examples first, we'll explain the differences afterwards. Here is a simple grammar and its usage in *Spirit.Classic*:
```cpp
struct roman : public grammar<roman>
{
    template <typename ScannerT>
    struct definition
    {
        definition(roman const& self)
        {
            hundreds.add
            ("C" , 100) ("CC" , 200) ("CCC" , 300) ("CD" , 400) ("D" , 500)
            ("DC" , 600) ("DCC" , 700) ("DCCC" , 800) ("CM" , 900);

            tens.add
            ("X" , 10) ("XX" , 20) ("XXX" , 30) ("XL" , 40) ("L" , 50)
            ("LX" , 60) ("LXX" , 70) ("LXXX" , 80) ("XC" , 90);

            ones.add
            ("I" , 1) ("II" , 2) ("III" , 3) ("IV" , 4) ("V" , 5)
            ("VI" , 6) ("VII" , 7) ("VIII" , 8) ("IX" , 9);

            first = eps_p
            [phoenix::var(self.r) = phoenix::val(0)]
            >> ( +ch_p('M') [phoenix::var(self.r) += phoenix::val(1000)]
                | hundreds [phoenix::var(self.r) += phoenix::_1]
                | tens [phoenix::var(self.r) += phoenix::_1]
                | ones [phoenix::var(self.r) += phoenix::_1]
            ) ;

        }
    }

    rule<ScannerT> first;  
    symbols<unsigned> hundreds; 
    symbols<unsigned> tens; 
    symbols<unsigned> ones; 

    rule<ScannerT> const& start() const { return first; }
};

roman(unsigned& r_) : r(r_) {}

std::string input("MMIX"); // MMIX == 2009
unsigned value = 0;
roman r(value);
parse_info<std::string::iterator> pi = parse(input.begin(), input.end(), r);
if (pi.hit)
    std::cout << "successfully matched: " << value << "\n";
```

And here is a similar grammar and its usage in Spirit.Qi:

```cpp
```
template <typename Iterator>
struct roman : qi::grammar<Iterator, unsigned()>
{
    roman() : roman::base_type(first)
    {
        hundreds.add
        ("C" , 100) ("CC" , 200) ("CCC" , 300) ("CD" , 400) ("D" , 500)
        ("DC" , 600) ("DCC" , 700) ("DCCC" , 800) ("CM" , 900) ;

        tens.add
        ("X" , 10) ("XX" , 20) ("XXX" , 30) ("XL" , 40) ("L" , 50)
        ("LX" , 60) ("LXX" , 70) ("LXXX" , 80) ("XC" , 90) ;

        ones.add
        ("I" , 1) ("II" , 2) ("III" , 3) ("IV" , 4) ("V" , 5)
        ("VI" , 6) ("VII" , 7) ("VIII" , 8) ("IX" , 9) ;

        // qi::_val refers to the attribute of the rule on the left hand side
        first = eps [qi::_val = 0]
        >> ( +lit('M') [qi::_val += 1000]
           | hundreds [qi::_val += qi::_1]
           | tens [qi::_val += qi::_1]
           | ones [qi::_val += qi::_1]
        ) ;

        qi::rule<Iterator, unsigned()> first;
        qi::symbols<char, unsigned> hundreds;
        qi::symbols<char, unsigned> tens;
        qi::symbols<char, unsigned> ones;
    }
};

std::string input("MMIX");       // MMIX == 2009
std::string::iterator it = input.begin();
unsigned value = 0;
roman<std::string::iterator> r;
if (qi::parse(it, input.end(), r, value))
    std::cout << "successfully matched: " << value << "\n";

Both versions look similar enough, but we see several differences (we will cover each of those differences in more detail below):

- Neither the grammars nor the rules depend on a scanner type anymore, both depend only on the underlying iterator type. That means the dreaded scanner business is no issue anymore!
- Grammars have no embedded class definition anymore
- Grammars and rules may have an explicit attribute type specified in their definition
- Grammars do not have any explicit start rules anymore. Instead one of the contained rules is used as a start rule by default.

The first two points are tightly interrelated. The scanner business (see the FAQ number one of Spirit.Classic here: The Scanner Business) has been a problem for a long time. The grammar and rule types have been specifically redesigned to avoid this problem in the future. This also means that we don't need any delayed instantiation of the inner definition class in a grammar anymore. So the redesign not only helped fixing a long standing design problem, it helped to simplify things considerably.

All Spirit.Qi parser components have well defined attribute types. Grammars and rules are no exception. But since both need to be generic enough to be usable for any parser their attribute type has to be explicitly specified. In the example above the roman grammar and the rule first both have an unsigned attribute:
The used notation resembles the definition of a function type. This is very natural as you can think of the synthesized attribute of the grammar and the rule as of its 'return value'. In fact the rule and the grammar both 'return' an unsigned value - the value they matched.

**Note**

The function type notation allows to specify parameters as well. These are interpreted as the types of inherited attributes the rule or grammar expect to be passed during parsing. For more information please see the section about inherited and synthesized attributes for rules and grammars ([Attributes](#)).

If no attribute is desired none needs to be specified. The default attribute type for both, grammars and rules, is `unused_type`, which is a special placeholder type. Generally, using `unused_type` as the attribute of a parser is interpreted as 'this parser has no attribute'. This is mostly used for parsers applied to parts of the input not carrying any significant information, rather being delimiters or structural elements needed for correct interpretation of the input.

The last difference might seem to be rather cosmetic and insignificant. But it turns out that not having to specify which rule in a grammar is the start rule (by returning it from the function `start()` also means that any rule in a grammar can be directly used as the start rule. Nevertheless, the grammar base class gets initialized with the rule it has to use as the start rule in case the grammar instance is directly used as a parser.

### Style Guide

At some point, especially when there are lots of semantic actions attached to various points, the grammar tends to be quite difficult to follow. In order to keep an easy-to-read, consistent and aesthetically pleasing look to the Spirit code, the following coding style guide is advised.

This coding style is adapted and extended from the ANTLR/PCCTS style and [Boost Library Requirements and Guidelines](#) and is the combined work of Joel de Guzman, Chris Uzdavinis, and Hartmut Kaiser.

- Rule names use std C++ (Boost) convention. The rule name may be very long.
- The '=' is neatly indented 4 spaces below. Like in Boost, use spaces instead of tabs.
- Breaking the operands into separate lines puts the semantic actions neatly to the right.
- Semicolon at the last line terminates the rule.
- The adjacent parts of a sequence should be indented accordingly to have all, what belongs to one level, at one indentation level.

```cpp
program
  =  program_heading [heading_action]
  >> block [block_action]
  >> '.'
  | another_sequence
  >> etc;
```

- Prefer literals in the grammar instead of identifiers. e.g. "program" instead of `PROGRAM`, '>=-' instead of `GTE` and '.,' instead of `DOT`. This makes it much easier to read. If this isn't possible (for instance where the used tokens must be identified through integers) capitalized identifiers should be used instead.
• Breaking the operands may not be needed for short expressions. e.g. `*'(,)' >> file_identifier` as long as the line does not exceed 80 characters.

• If a sequence fits on one line, put spaces inside the parentheses to clearly separate them from the rules.

```spirit
program_heading
  = no_case("program")
    >> identifier
    >> '('
    >> file_identifier
    >> *'(,)' >> file_identifier )
    >> ')
    >> ';'
  ;
```

• Nesting directives: If a rule does not fit on one line (80 characters) it should be continued on the next line intended by one level. The brackets of directives, semantic expressions (using Phoenix or LL lambda expressions) or parsers should be placed as follows.

```spirit
identifier
  = no_case
    [ lexeme
      [ alpha >> *(alnum | '_') [id_action] ]
    ]
  ;
```

• Nesting unary operators (e.g. Kleene star): Unary rule operators (Kleene star, '!', '+' etc.) should be moved out one space before the corresponding indentation level, if this rule has a body or a sequence after it, which does not fit on one line. This makes the formatting more consistent and moves the rule 'body' at the same indentation level as the rule itself, highlighting the unary operator.

```spirit
block
  = *
    [ label_declaration_part
      | constant_definition_part
      | type_definition_part
      | variable_declaration_part
      | procedure_and_function_declaration_part
    ]
    >> statement_part
  ;
```
Rationale

Naming

Why use the name "Spirit", "Qi" and "Karma"? Because xpressive names have a better spirit, brings qi to your software and will enhance your karma so they can heal your (con)fusion and make you wave like a phoenix from the ashes. (Joachim Faulhaber)

Type Erasure: From static to dynamic C++

Rules straddle the border between static and dynamic C++. In effect, a rule transforms compile-time polymorphism (using templates) into run-time polymorphism (using virtual functions). This is necessary due to C++'s inability to automatically declare a variable of a type deduced from an arbitrarily complex expression in the right-hand side (rhs) of an assignment. Basically, we want to do something like:

```cpp
T rule = an_arbitrarily_complex_expression;
```

without having to know or care about the resulting type of the right-hand side (rhs) of the assignment expression. This can be done with modern C++ 0x compilers using `auto`:

```cpp
auto rule = an_arbitrarily_complex_expression;
```

Apart from this, we also need a facility to forward declare an unknown type:

```cpp
T rule;
...
rule = a | b;
```

These limitations lead us to this implementation of rules. This comes at the expense of the overhead of a type-erased call which is an indirect function call that cannot be inlined, once through each invocation of a rule.

Multiple declaration

Some BNF variants allow multiple declarations of a rule. The declarations are taken as alternatives. Example:

```cpp
r = a;
r = b;
```

is equivalent to:

```cpp
r = a | b;
```

Spirit v1.3 allowed this behavior. However, the current version of Spirit no longer allows this because experience shows that this behavior leads to unwanted gotchas (for instance, it does not allow rules to be held in containers). In the current release of Spirit, a second assignment to a rule will simply redefine it. The old definition is destructed. This follows more closely C++ semantics and is more in line with what the user expects the rule to behave.

Sequencing Syntax

The comma operator as in `a, b` seems to be a better candidate, syntax-wise. But then the problem is with its precedence. It has the lowest precedence in C/C++, which makes it virtually useless.

Bjarne Stroustrup, in his article "Generalizing Overloading for C++2000" talks about overloading whitespace. Such a feature would allow juxtapositioning of parser objects exactly as we do in (E)BNF (e.g. a b | c instead of a >> b | c). Unfortunately, the article was dated April 1, 1998. Oh well.
Forward iterators

In general, the expected iterator is at least a standard conforming forward iterator. Forward iterators are needed for backtracking where the iterator needs to be saved and restored later. Generally speaking, Spirit is a backtracking parser. The implication of this is that at some point, the iterator position will have to be saved to allow the parser to backtrack to a previous point. Thus, for backtracking to work, the framework requires at least a forward iterator.

There are remedies of course. In cases where we need to use input iterators, you can use the `multi_pass` iterator to make the forward iterators.

Some parsers might require more specialized iterators (bi-directional or even random access). Perhaps in the future, deterministic parsers when added to the framework, will perform no backtracking and will need just a single token lookahead, hence will require input iterators only.

Exhaustive backtracking and greedy RD

Spirit doesn't do exhaustive backtracking like regular expressions are expected to. For example:

```
*char_('a') >> char_('a');
```

will always fail to match because Spirit's Kleene star does not back off when the rest of the rule fails to match.

Actually, there's a solution to this greedy RD problem. Such a scheme is discussed in section 6.6.2 of Parsing Techniques: A Practical Guide. The trick involves passing a tail parser (in addition to the scanner) to each parser. The start parser will then simply be:

```
start >> end;
```

(end is the start's tail).

Spirit is greedy --using straight forward, naive RD. It is certainly possible to implement the fully backtracking scheme presented above, but there will be also certainly be a performance hit. The scheme will always try to match all possible parser paths (full parser hierarchy traversal) until it reaches a point of certainty, that the whole thing matches or fails to match.

Backtracking and Greedy RD

Spirit is quite consistent and intuitive about when it backtracks and to where, although it may not be obvious to those coming from different backgrounds. In general, any (sub)parser will, given the same input, always match the same portion of the input (or fail to match the input at all). This means that Spirit is inherently greedy. Spirit will only backtrack when a (sub)parser fails to match the input, and it will always backtrack to the next choice point upward (not backward) in the parser structure. In other words `abbr` will match "ab", as will `a bb| b`, but `ab| a | b` won't because the `ab| a` subparser will always match the 'b' after the 'a' if it is available.

--Rainer Deyke

This is the very nature of Parsing Expression Grammar.

There's a strong preference on "simplicity with all the knobs when you need them" approach, right now. On the other hand, the flexibility of Spirit makes it possible to have different optional schemes available. It might be possible to implement an exhaustive backtracking RD scheme as an optional feature in the future.
Spirit Repository

The Spirit repository is a community effort collecting different reusable components (primitives, directives, grammars, etc.) for Spirit.Qi parsers and Spirit.Karma generators. All components in the repository have been peer reviewed or discussed on the Spirit General List. For more information about the repository see here.
Acknowledgments

This version of Spirit is a complete rewrite of the classic Spirit many people have been contributing to (see below). But there are a couple of people who already managed to help significantly during this rewrite. We would like to express our special acknowledgement to (in no particular order):

**Eric Niebler** for writing Boost.Proto, without which this rewrite wouldn't have been possible, and helping with examples, advice, and suggestions on how to use Boost.Proto in the best possible way.

**Ben Hanson** for providing us with an early version of his Lexertl library, which is proposed to be included into Boost (as Boost.Lexer). At the time of this writing the Boost review for this library is still pending.

**Francois Barel** for his silent but steady work on making and keeping Spirit compatible with all versions of gcc, older and newest ones. He not only contributed subrules to Spirit V2.1, but always keeps an eye on the small details which are so important to make a difference.

**Andreas Haberstroh** for proof reading the documentation and fixing those non-native-speaker-quirks we managed to introduce into the first versions of the documentation.

**Chris Hoeppler** for taking up the editorial tasks for the initial version of this documentation together with Andreas Haberstroh. Chris did a lot especially at the last minute when we are about to release.

**Michael Caisse** (a.k.a mjcaisse) for being Spirit's benevolent evangelist on freenode's #boost IRC. No Spirit question asked there stays without an answer because of his active involvement. Also, we would like to thank him for last minute editing work on the 2.1 release documentation.

**Tobias Schwinger** for proposing expectation points and GCC port of an early version.

**Dave Abrahams** as always, for countless advice and help on C++, library development, interfaces, usability and ease of use, for reviewing the code and providing valuable feedback and for always keeping us on our toes.

**OvermindDL** for his creative ideas on the mailing list helping to resolve even more difficult user problems.

**Carl Barron** for his early adoption and valuable feedback on the Lexer library forcing us to design a proper API covering all of his use cases. He also contributed an early version of the variadic attribute API for Qi.

**Daniel James** for improving and maintaining Quickbook, the tool we use for this documentation. Also, for bits and pieces here and there such documentation suggestions and editorial patches.

**Stephan Menzel** for his early adoption of Qi and Karma and his willingness to invest time to spot bugs which were hard to isolate. Also, for his feedback on the documentation.

**Ray Burkholder** and **Dainis Polis** for last minute feedback on the documentation.

**Steve Brandt** for his his effort trying to put Qi and Karma to some use while writing a source to source language transformation tool. He made many incredible suggestions helping to improve the usability of both libraries.

**Bryce Leylbach** (a.k.a. wash) for contributing the components qi::as<T>[] and karma::as<T>[], for working on utree and its integration with Qi and Karma, for adding numeric parser literals, for contributing the related documentation, and for being a big help in the everyday maintenance of the library's (and our) sanity.

**Mathias Gaunard** for his bug reports, suggestions and regression test contributions to the Lexer helping to find subtle bugs and to improve its overall usability.

**Thomas Bernard** (a.k.a. teajay) for working on the keyword parser and for adding explicit names to the qi::symbols<> component.

**Robert Stewart** for his active participation on the mailing list, his helpful comments, examples and suggestions.

**Thomas Taylor**, **Richard Crossley**, **Semen**, and **Adalberto Castelo** for their help in isolating problems while testing the new attribute handling code released with V2.5.
Special thanks to spirit-devel and spirit-general mailing lists for participating in the discussions, being early adopters of pre-release versions of Spirit2 from the very start and helping out in various tasks such as helping with support, bug tracking, benchmarking and testing, etc. The list include: Larry Evans, Richard Webb, Martin Wille, Dan Marsden, Cedric Venet, Allan Odgaard, Matthias Vallentin, Justinas V.D., Darid Tromer, Brian O'Kennedy, Aaron Graham, Joerg Becker.

Joao Abecasis for his early support and involvement in Spirit2 development and for disturbing my peace every once in a while for a couple of jokes.

The list goes on and on... if you've been mentioned thank Joel and Hartmut, if not, kick Joao :-)

**Acknowledgements from the Spirit V1 classic Documentation**

Special thanks for working on Spirit classic to:

Dan Nuffer for his work on lexers, parse trees, ASTs, XML parsers, the multi-pass iterator as well as administering Spirit's site, editing, maintaining the CVS and doing the releases plus a zillion of other chores that were almost taken for granted.

Hartmut Kaiser for his work on the C parser, the work on the C/C++ preprocessor, utility parsers, the original port to Intel 5.0, various work on Phoenix, porting to v1.5, the meta-parsers, the grouping-parsers, extensive testing and painstaking attention to details.

Martin Wille who improved grammar multi thread safety, contributed the eol_p parser, the dynamic parsers, documentation and for taking an active role in almost every aspect from brainstorming and design to coding. And, as always, helps keep the regression tests for g++ on Linux as green as ever :-).

Martijn W. Van Der Lee our Web site administrator and for contributing the RFC821 parser.

Giovanni Bajo for last minute tweaks of Spirit 1.8.0 for CodeWarrior 8.3. Actually, I'm ashamed Giovanni was not in this list already. He's done a lot since Spirit 1.5, the first Boost.Spirit release. He's instrumental in the porting of the Spirit iterators stuff to the new Boost Iterators Library (version 2). He also did various bug fixes and wrote some tests here and there.

Juan Carlos Arevalo-Baeza (JCAB)* for his work on the C++ parser, the position iterator, ports to v1.5 and keeping the mailing list discussions alive and kicking.

Vaclav Vesely, lots of stuff, the no_actions directive, various patches fixes, the distinct parsers, the lazy parser, some phoenix tweaks and add-ons (e.g. new_). Also, *Stefan Slapeta] and wife for editing Vaclav's distinct parser doc.

Raghadendra Satish for doing the original v1.3 port to VC++ and his work on Phoenix.

Noah Stein for following up and helping Ragav on the VC++ ports.

Hakki Dogusan, for his original v1.0 Pascal parser.

John (EB0) David for his work on the VM and watching over my shoulder as I code giving the impression of distance eXtreme programming.

Chris Uzdavinis for feeding in comments and valuable suggestions as well as editing the documentation.

Carsten Stoll, for his work on dynamic parsers.

Andy Elvey and his conifer parser.

Bruce Florman, who did the original v1.0 port to VC++.

Jeff Westfahl for porting the loop parsers to v1.5 and contributing the file iterator.

Peter Simons for the RFC date parser example and tutorial plus helping out with some nitty gritty details.

Markus Schöpflin for suggesting the end_p parser and lots of other nifty things and his active presence in the mailing list.

Doug Gregor for mentoring and his ability to see things that others don't.
David Abrahams for giving Joel a job that allows him to still work on Spirit, plus countless advice and help on C++ and specifically template metaprogramming.

Aleksey Gurtovoy for his MPL library from which we stole many metaprogramming tricks especially for less conforming compilers such as Borland and VC6/7.

Gustavo Guerra for his last minute review of Spirit and constant feedback, plus patches here and there (e.g. proposing the new dot behavior of the real numerics parsers).

Nicola Musatti, Paul Snively, Alisdair Meredith and Hugo Duncan for testing and sending in various patches.

Steve Rowe for his splendid work on the TSTs that will soon be taken into Spirit.

Jonathan de Halleux for his work on actors.

Angus Leeming for last minute editing work on the 1.8.0 release documentation, his work on Phoenix and his active presence in the Spirit mailing list.

Joao Abecasis for his active presence in the Spirit mailing list, providing user support, participating in the discussions and so on.

Guillaume Melquiond for a last minute patch to multi_pass for 1.8.1.

Peder Holt for his porting work on Phoenix, Fusion and Spirit to VC6.

To Joel's wife Mariel who did the graphics in this document.

My, there's a lot in this list! And it's a continuing list. We add people to this list every time. We hope we did not forget anyone. If we missed someone you know who has helped in any way, please inform us.

Special thanks also to people who gave feedback and valuable comments, particularly members of Boost and Spirit mailing lists. This includes all those who participated in the review:

John Maddock, our review manager, Aleksey Gurtovoy, Andre Hentz, Beman Dawes, Carl Daniel, Christopher Currie, Dan Gohman, Dan Nuffer, Daryle Walker, David Abrahams, David B. Held, Dirk Gerrits, Douglas Gregor, Hartmut Kaiser, Iain K.Hanson, Juan Carlos Arevalo-Baeza, Larry Evans, Martin Wille, Mattias Flodin, Noah Stein, Nuno Lucas, Peter Dimov, Peter Simons, Petr Kocmrd, Ross Smith, Scott Kirkwood, Steve Cleary, Thorsten Ottosen, Tom Wenisch, Vladimir Prus

Finally thanks to SourceForge for hosting the Spirit project and Boost: a C++ community comprised of extremely talented library authors who participate in the discussion and peer review of well crafted C++ libraries.
References
<table>
<thead>
<tr>
<th>Authors</th>
<th>Title, Publisher/link, Date Published</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geoffrey Furnish</td>
<td>&quot;Disambiguated Glommable Expression Templates Reintroduced&quot; C++ Report, May 2000</td>
</tr>
<tr>
<td>Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides</td>
<td>Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.</td>
</tr>
<tr>
<td>Adrian Johnstone</td>
<td>Languages and Architectures, Parser generators with backtrack or extended lookahead capability. Department of Computer Science, Royal Holloway, University of London, Egham, Surrey, England</td>
</tr>
<tr>
<td>Authors</td>
<td>Title, Publisher/link, Date Published</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Joel de Guzman</td>
<td>&quot;Spirit Version 1.8&quot;, 1998-2003</td>
</tr>
<tr>
<td>S. Doaitse Swierstra and Luc Duponcheel</td>
<td>Deterministic, Error-Correcting Combinator Parsers Dept. of Computer Science, Utrecht University P.O.Box 80.089, 3508 TB Utrecht, The Netherland</td>
</tr>
<tr>
<td>Dr. John Maddock</td>
<td>Regex++ Documentation <a href="http://www.boost.org/libs/regex/index.htm">http://www.boost.org/libs/regex/index.htm</a></td>
</tr>
<tr>
<td>Brian McNamara and Yannis Smaragdakis</td>
<td>FC++:Functional Programming in C++.</td>
</tr>
<tr>
<td>Todd Veldhuizen</td>
<td>Techniques for Scientific C++.</td>
</tr>
</tbody>
</table>