# Boost Spirit API

D. Gueorguiev 10/9/21

# Main classes and functionality

## Tokenization

### The tokenize() function

The [tokenize()](https://github.com/boostorg/spirit/blob/boost-1.75.0/include/boost/spirit/home/lex/tokenize_and_parse.hpp#L227-L327) function is one of the main Spirit API functions. It resides entirely in the header [tokenize\_and\_parse.hpp](https://github.com/boostorg/spirit/blob/boost-1.75.0/include/boost/spirit/home/lex/tokenize_and_parse.hpp).

It simplifies the usage of lexer to tokenize a given input sequence. Its main purpose is to use the lexer to tokenize all the input.

The second version of [tokenize()](https://github.com/boostorg/spirit/blob/boost-1.75.0/include/boost/spirit/home/lex/tokenize_and_parse.hpp#L313-L327) (shown below) discards all generated tokens afterwards. This is useful whenever all the needed functionality has been implemented directly inside the lexer semantic actions, which are being executed while the tokens are matched.

The function takes a pair of iterators spanning the underlying stream to scan, the lexer object built from the token definitions, and an optional functor being called for each of the generated tokens.

The function returns true if the scanning of the input succeeded which is the given input sequence has been successfully matched by the given token definitions. Here are the arguments:

first, last: The pair of iterators spanning the underlying input sequence to parse. These iterators must at least conform to the requirements of [std::input\_iterator](https://en.cppreference.com/w/cpp/iterator/input_iterator) category.

Side note 1:

The base concept of std::input\_iterator is [std::input\_or\_output\_iterator](https://en.cppreference.com/w/cpp/iterator/input_or_output_iterator) (defined in header [<iterator>](https://en.cppreference.com/w/cpp/header/iterator)):

template <class I>

concept input\_or\_output\_iterator =

requires(I i) {

{ \*i } -> \_\_Referenceable;

} &&

std::weakly\_incrementable<I>;

The input\_or\_output\_iterator concept forms the basis of the iterator concept taxonomy; every iterator type satisfies the

input\_or\_output\_iterator requirements. The exposition-only concept \_\_Referenceable if and only if the expression \*std::declval<I&>() is valid and has referenceable type (in particular not void).

Side node 2*:*

*Equality Preservation*

An expression is *equality preserving* if it results in equal outputs given equal inputs.

The inputs to an expression consists of its operands

The outputs of an expression consists of its result and all operands modified by the expression (if any).

In specification of standard concepts, operands are defined as the largest subexpressions that include only:

* an id-expression *and*
* invocations of std::move, std::forward and std::declval

The cv-qualification and value category of each operand is determined by assuming that each template type parameter denotes cv-unqualified complete non-array object type.

Every expression required to be equality-preserving is further required to be stable: two evaluations of such an expression with the same input objects must have equal outputs absent any explicit any explicit intervening modification of those input objects.

Unless noted otherwise, every expression used in a *requires-expression* is required to be equality preserving and stable, and the evaluation of the expression may only modify its non-constant operands. Operands that are constant must not be modified.

Side note 3:

[std::weakly\_incrementable](https://en.cppreference.com/w/cpp/iterator/weakly_incrementable)

template<class I>

concept weakly\_incrementable =

std::movable<I> &&

requires(I i) {

typename std::iter\_difference\_t<I>;

requires /\* is-signed-integer-like \*/ <std::iter\_difference\_t<I>>;

{ ++i } -> std::same\_as<I&>; // not required to be equality-preserving

i++; // not required to be equality-preserving

};

where /\* is-signed-integer-like \*/ <std::iter\_difference\_t<I>> is true if and only if I is a signed integer-like type (see below).

This concept specifies requirements on types that can be incremented with the pre- and post-increment operators, but those increment operators are not necessarily equality-preserving, and the type itself is not required to be [std::equality\_comparable](https://en.cppreference.com/w/cpp/concepts/equality_comparable).

For std::weakly\_incrementable types, a == b does not imply ++a == ++b. Algorithms on weakly incrementable types must be single pass algorithms. These algorithms can be used with istreams as the source of the input data through std::input\_iterator.

Semantic requirements

I models std::weakly\_incrementable only if given only object i of type I:

* the expressions ++i and i++ have the same domain
* if i is incrementable, then both ++i and i++ advance i
* if i is incrementable, then std::addressof(++i) == std::addressof(i)

Integer-like types

An integer-like type is an (possibly cv-qualified) integer type (except for cv bool) or an implementation-provided (not user-provided) class that behaves like an integer type, including all operators, implicit conversions, [std::numeric\_limits](https://en.cppreference.com/w/cpp/types/numeric_limits) specializations. If an integer-like type only represents non-negative values, it is unsigned-integer-like, otherwise it is signed-integer-like.

Side note 4:

std::indirectly\_readable

template<class In>

concept \_\_IndirectlyReadableImpl = // exposition only

requires(const In in) {

typename std::iter\_value\_t<In>;

typename std::iter\_reference\_t<In>;

typename std::iter\_rvalue\_reference<In>;

{ \*in } -> std::same\_as<std::iter\_reference\_t<In>>;

{ ranges::iter\_move(in) } -> std::same\_as<std::iter\_rvalue\_reference\_t<In>>;

} &&

std::common\_reference\_wth<

TODO: finish this

Side note 5:

[std::common\_reference](https://en.cppreference.com/w/cpp/types/common_reference)

template< class... T >

struct common\_reference;

Determines the common reference type of the types T... that is the type to which all of the types in T... can be converted or bound. If such a type exist (as determined according to the rules below) , the member std::common\_reference<T...>::type names that type. Otherwise, there is no member std::common\_reference<T...>::type. The behavior is undefined if any of the types in T... is incomplete type other than (possibly cv-qualified) void.

When given reference types, common\_reference attempts to find a reference type to which the supplied reference types can all be bound, but may return a non-reference type if it cannot find such a reference type.

* if