Boost.Xpressive

Eric Niebler

Table of Contents

Preface

Wife: New Shimmer is a floor wax!
Husband: No, new Shimmer is a dessert topping!
Wife: It's a floor wax!
Husband: It's a dessert topping!
Wife: It's a floor wax, I'm telling you!
Husband: It's a dessert topping, you cow!
Announcer: Hey, hey, hey, calm down, you two. New Shimmer is both a floor wax *and* a dessert topping!

-- Saturday Night Live

Description

xpressive is an advanced, object-oriented regular expression template library for C++. Regular expressions can be written as strings that are parsed at run-time, or as expression templates that are parsed at compile-time. Regular expressions can refer to each other and to themselves recursively, allowing you to build arbitrarily complicated grammars out of them.

Motivation

If you need to manipulate text in C++, you have typically had two disjoint options: a regular expression engine or a parser generator. Regular expression engines (like Boost.Regex) are powerful and flexible; patterns are represented as strings which can be specified at runtime. However, that means that syntax errors are likewise not detected until runtime. Also, regular expressions are ill-suited to advanced text processing tasks such as matching balanced, nested tags. Those tasks have traditionally been handled by parser generators (like the Spirit Parser Framework). These beasts are more powerful but less flexible. They generally don't allow you to arbitrarily modify your grammar rules on the fly. In addition, they don't have the exhaustive backtracking semantics of regular expressions, which can make it more challenging to author some types of patterns.

xpressive brings these two approaches seamlessly together and occupies a unique niche in the world of C++ text processing. With xpressive, you can choose to use it much as you would use Boost.Regex, representing regular expressions as strings. Or you can use it as you would use Spirit, writing your regexes as C++ expressions, enjoying all the benefits of an embedded language dedicated to text manipulation. What's more, you can mix the two to get the benefits of both, writing regular expression *grammars* in which some of the regular expressions are statically bound -- hard-coded and syntax-checked by the compiler -- and others are dynamically bound and specified at runtime. These regular expressions can refer to each other recursively, matching patterns in strings that ordinary regular expressions cannot.

Influences and Related Work

The design of xpressive's interface has been strongly influenced by John Maddock's Boost.Regex library and his proposal to add regular expressions to the Standard Library. I also drew a great deal of inspiration from Joel de Guzman's Spirit Parser Framework, which served as the model for static xpressive. Other sources of inspiration are the Perl 6 redesign and GRETA. (You can read a summary of the changes Perl 6 will bring to regex culture here.)

User's Guide

This section describes how to use xpressive to accomplish text manipulation and parsing tasks. If you are looking for detailed information regarding specific components in xpressive, check the Reference section.

Introduction

What is xpressive?

xpressive is a regular expression template library. Regular expressions (regexes) can be written as strings that are parsed dynamically at runtime (dynamic regexes), or as *expression templates*[1] that are parsed at compile-time (static regexes). Dynamic regexes have the advantage that they can be accepted from the user as input at runtime or read from an initialization file. Static regexes have several advantages. Since they are C++ expressions instead of strings, they can be syntax-checked at compile-time. Also, they can naturally refer to code and data elsewhere in your program, giving you the ability to call back into your code from within a regex match. Finally, since they are statically bound, the compiler can generate faster code for static regexes.

xpressive's dual nature is unique and powerful. Static xpressive is a bit like the Spirit Parser Framework. Like Spirit, you can build grammars with static regexes using expression templates. (Unlike Spirit, xpressive does exhaustive backtracking, trying every possibility to find a match for your pattern.) Dynamic xpressive is a bit like Boost.Regex. In fact, xpressive's interface should be familiar to anyone who has used Boost.Regex. xpressive's innovation comes from allowing you to mix and match static and dynamic regexes in the same program, and even in the same expression! You can embed a dynamic regex in a static regex, or *vice versa*, and the embedded regex will participate fully in the search, back-tracking as needed to make the match succeed.

Hello, world!

Enough theory. Let's have a look at *Hello World*, xpressive style:

```
#include <iostream>
#include <boost/xpressive/xpressive.hpp>

using namespace boost::xpressive;

int main()
{
    std::string hello( "hello world!" );

    sregex rex = sregex::compile( "(\\w+) (\\w+)!" );
    smatch what;

    if( regex_match( hello, what, rex ) )
    {
        std::cout << what[0] << '\n'; // whole match
        std::cout << what[1] << '\n'; // first capture
        std::cout << what[2] << '\n'; // second capture
    }

    return 0;
}
```

This program outputs the following:

```
hello world!
hello
world
```

[1] See Expression Templates

The first thing you'll notice about the code is that all the types in xpressive live in the `boost::xpressive` namespace.

Note

Most of the rest of the examples in this document will leave off the `using namespace boost::xpressive;` directive. Just pretend it's there.

Next, you'll notice the type of the regular expression object is `sregex`. If you are familiar with Boost.Regex, this is different than what you are used to. The "`s`" in "`sregex`" stands for "`string`", indicating that this regex can be used to find patterns in `std::string` objects. I'll discuss this difference and its implications in detail later.

Notice how the regex object is initialized:

```
sregex rex = sregex::compile( "(\\w+) (\\w+)!" );
```

To create a regular expression object from a string, you must call a factory method such as `basic_regex<>::compile()`. This is another area in which xpressive differs from other object-oriented regular expression libraries. Other libraries encourage you to think of a regular expression as a kind of string on steroids. In xpressive, regular expressions are not strings; they are little programs in a domain-specific language. Strings are only one *representation* of that language. Another representation is an expression template. For example, the above line of code is equivalent to the following:

```
sregex rex = (s1= +_w) >> ' ' >> (s2= +_w) >> '!';
```

This describes the same regular expression, except it uses the domain-specific embedded language defined by static xpressive.

As you can see, static regexes have a syntax that is noticeably different than standard Perl syntax. That is because we are constrained by C++'s syntax. The biggest difference is the use of >> to mean "followed by". For instance, in Perl you can just put sub-expressions next to each other:

```
abc
```

But in C++, there must be an operator separating sub-expressions:

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

In Perl, parentheses `( )` have special meaning. They group, but as a side-effect they also create back-references like `$1` and `$2`. In C++, there is no way to overload parentheses to give them side-effects. To get the same effect, we use the special `s1`, `s2`, etc. tokens. Assign to one to create a back-reference (known as a sub-match in xpressive).

You'll also notice that the one-or-more repetition operator `+` has moved from postfix to prefix position. That's because C++ doesn't have a postfix `+` operator. So:

```
"\\w+"
```

is the same as:

```
+_w
```

We'll cover all the other differences later.

Installing xpressive

Getting xpressive

There are two ways to get xpressive. The first and simplest is to download the latest version of Boost. Just go to http://sf.net/projects/boost and follow the “Download” link.

The second way is by directly accessing the Boost Subversion repository. Just go to http://svn.boost.org/trac/boost/ and follow the instructions there for anonymous Subversion access. The version in Boost Subversion is unstable.

Building with xpressive

Xpressive is a header-only template library, which means you don't need to alter your build scripts or link to any separate lib file to use it. All you need to do is `#include <boost/xpressive/xpressive.hpp>`. If you are only using static regexes, you can improve compile times by only including `xpressive_static.hpp`. Likewise, you can include `xpressive_dynamic.hpp` if you only plan on using dynamic regexes.

If you would also like to use semantic actions or custom assertions with your static regexes, you will need to additionally include `regex_actions.hpp`.

Requirements

Xpressive requires Boost version 1.34.1 or higher.

Supported Compilers

Currently, Boost.Xpressive is known to work on the following compilers:

- Visual C++ 7.1 and higher
- GNU C++ 3.4 and higher
- Intel for Linux 8.1 and higher
- Intel for Windows 10 and higher
- tru64cxx 71 and higher
- MinGW 3.4 and higher
- HP C/aC++ A.06.14

Check the latest tests results at Boost's Regression Results Page.

Note

Please send any questions, comments and bug reports to eric <at> boost-consulting <dot> com.

Quick Start

You don't need to know much to start being productive with xpressive. Let's begin with the nickel tour of the types and algorithms xpressive provides.

Table 1. xpressive's Tool-Box

Tool	Description
`basic_regex<>`	Contains a compiled regular expression. `basic_regex<>` is the most important type in xpressive. Everything you do with xpressive will begin with creating an object of type `basic_regex<>`.
`match_results<>`, `sub_match<>`	`match_results<>` contains the results of a `regex_match()` or `regex_search()` operation. It acts like a vector of `sub_match<>` objects. A `sub_match<>` object contains a marked sub-expression (also known as a back-reference in Perl). It is basically just a pair of iterators representing the begin and end of the marked sub-expression.
`regex_match()`	Checks to see if a string matches a regex. For `regex_match()` to succeed, the *whole string* must match the regex, from beginning to end. If you give `regex_match()` a `match_results<>`, it will write into it any marked sub-expressions it finds.
`regex_search()`	Searches a string to find a sub-string that matches the regex. `regex_search()` will try to find a match at every position in the string, starting at the beginning, and stopping when it finds a match or when the string is exhausted. As with `regex_match()`, if you give `regex_search()` a `match_results<>`, it will write into it any marked sub-expressions it finds.
`regex_replace()`	Given an input string, a regex, and a substitution string, `regex_replace()` builds a new string by replacing those parts of the input string that match the regex with the substitution string. The substitution string can contain references to marked sub-expressions.
`regex_iterator<>`	An STL-compatible iterator that makes it easy to find all the places in a string that match a regex. Dereferencing a `regex_iterator<>` returns a `match_results<>`. Incrementing a `regex_iterator<>` finds the next match.
`regex_token_iterator<>`	Like `regex_iterator<>`, except dereferencing a `regex_token_iterator<>` returns a string. By default, it will return the whole sub-string that the regex matched, but it can be configured to return any or all of the marked sub-expressions one at a time, or even the parts of the string that *didn't* match the regex.
`regex_compiler<>`	A factory for `basic_regex<>` objects. It "compiles" a string into a regular expression. You will not usually have to deal directly with `regex_compiler<>` because the `basic_regex<>` class has a factory method that uses `regex_compiler<>` internally. But if you need to do anything fancy like create a `basic_regex<>` object with a different `std::locale`, you will need to use a `regex_compiler<>` explicitly.

Now that you know a bit about the tools xpressive provides, you can pick the right tool for you by answering the following two questions:

1. What *iterator* type will you use to traverse your data?

2. What do you want to *do* to your data?

Know Your Iterator Type

Most of the classes in xpressive are templates that are parameterized on the iterator type. xpressive defines some common typedefs to make the job of choosing the right types easier. You can use the table below to find the right types based on the type of your iterator.

Table 2. xpressive Typedefs vs. Iterator Types

	std::string::const_iterator	**char const ***	**std::wstring::const_iterator**	**wchar_t const ***
`basic_regex<>`	`sregex`	`cregex`	`wsregex`	`wcregex`
`match_results<>`	`smatch`	`cmatch`	`wsmatch`	`wcmatch`
`regex_compiler<>`	`sregex_compiler`	`cregex_compiler`	`wsregex_compiler`	`wcregex_compiler`
`regex_iterator<>`	`sregex_iterator`	`cregex_iterator`	`wsregex_iterator`	`wcregex_iterator`
`regex_token_iterator<>`	`sregex_token_iterator`	`cregex_token_iterator`	`wsregex_token_iterator`	`wcregex_token_iterator`

You should notice the systematic naming convention. Many of these types are used together, so the naming convention helps you to use them consistently. For instance, if you have a `sregex`, you should also be using a `smatch`.

If you are not using one of those four iterator types, then you can use the templates directly and specify your iterator type.

Know Your Task

Do you want to find a pattern once? Many times? Search and replace? xpressive has tools for all that and more. Below is a quick reference:

Table 3. Tasks and Tools

To do this ...	**Use this ...**
See if a whole string matches a regex	The `regex_match()` algorithm
See if a string contains a sub-string that matches a regex	The `regex_search()` algorithm
Replace all sub-strings that match a regex	The `regex_replace()` algorithm
Find all the sub-strings that match a regex and step through them one at a time	The `regex_iterator<>` class
Split a string into tokens that each match a regex	The `regex_token_iterator<>` class
Split a string using a regex as a delimiter	The `regex_token_iterator<>` class

These algorithms and classes are described in excruciating detail in the Reference section.

Tip

Try clicking on a task in the table above to see a complete example program that uses xpressive to solve that particular task.

Creating a Regex Object

When using xpressive, the first thing you'll do is create a `basic_regex<>` object. This section goes over the nuts and bolts of building a regular expression in the two dialects xpressive supports: static and dynamic.

Static Regexes

Overview

The feature that really sets xpressive apart from other C/C++ regular expression libraries is the ability to author a regular expression using C++ expressions. xpressive achieves this through operator overloading, using a technique called *expression templates* to embed a mini-language dedicated to pattern matching within C++. These "static regexes" have many advantages over their string-based brethren. In particular, static regexes:

- are syntax-checked at compile-time; they will never fail at run-time due to a syntax error.
- can naturally refer to other C++ data and code, including other regexes, making it simple to build grammars out of regular expressions and bind user-defined actions that execute when parts of your regex match.
- are statically bound for better inlining and optimization. Static regexes require no state tables, virtual functions, byte-code or calls through function pointers that cannot be resolved at compile time.
- are not limited to searching for patterns in strings. You can declare a static regex that finds patterns in an array of integers, for instance.

Since we compose static regexes using C++ expressions, we are constrained by the rules for legal C++ expressions. Unfortunately, that means that "classic" regular expression syntax cannot always be mapped cleanly into C++. Rather, we map the regex *constructs*, picking new syntax that is legal C++.

Construction and Assignment

You create a static regex by assigning one to an object of type `basic_regex<>`. For instance, the following defines a regex that can be used to find patterns in objects of type `std::string`:

```
sregex re = '$' >> +_d >> '.' >> _d >> _d;
```

Assignment works similarly.

Character and String Literals

In static regexes, character and string literals match themselves. For instance, in the regex above, `'$'` and `'.'` match the characters `'$'` and `'.'` respectively. Don't be confused by the fact that `$` and `.` are meta-characters in Perl. In xpressive, literals always represent themselves.

When using literals in static regexes, you must take care that at least one operand is not a literal. For instance, the following are *not* valid regexes:

```
sregex re1 = 'a' >> 'b';          // ERROR!
sregex re2 = +'a';                // ERROR!
```

The two operands to the binary >> operator are both literals, and the operand of the unary + operator is also a literal, so these statements will call the native C++ binary right-shift and unary plus operators, respectively. That's not what we want. To get operator overloading to kick in, at least one operand must be a user-defined type. We can use xpressive's `as_xpr()` helper function to "taint" an expression with regex-ness, forcing operator overloading to find the correct operators. The two regexes above should be written as:

```
sregex re1 = as_xpr('a') >> 'b'; // OK
sregex re2 = +as_xpr('a');       // OK
```

Sequencing and Alternation

As you've probably already noticed, sub-expressions in static regexes must be separated by the sequencing operator, >>. You can read this operator as "followed by".

```
// Match an 'a' followed by a digit
sregex re = 'a' >> _d;
```

Alternation works just as it does in Perl with the | operator. You can read this operator as "or". For example:

```
// match a digit character or a word character one or more times
sregex re = +( _d | _w );
```

Grouping and Captures

In Perl, parentheses `()` have special meaning. They group, but as a side-effect they also create back-references like `$1` and `$2`. In C++, parentheses only group -- there is no way to give them side-effects. To get the same effect, we use the special `s1`, `s2`, etc. tokens. Assigning to one creates a back-reference. You can then use the back-reference later in your expression, like using `\1` and `\2` in Perl. For example, consider the following regex, which finds matching HTML tags:

```
"<(\\w+)>.*?</\\1>"
```

In static xpressive, this would be:

```
'<' >> (s1= +_w) >> '>' >> -*_ >> "</" >> s1 >> '>'
```

Notice how you capture a back-reference by assigning to `s1`, and then you use `s1` later in the pattern to find the matching end tag.

Tip

Grouping without capturing a back-reference
In xpressive, if you just want grouping without capturing a back-reference, you can just use `()` without `s1`. That is the equivalent of Perl's `(?:)` non-capturing grouping construct.

Case-Insensitivity and Internationalization

Perl lets you make part of your regular expression case-insensitive by using the `(?i:)` pattern modifier. xpressive also has a case-insensitivity pattern modifier, called `icase`. You can use it as follows:

```
sregex re = "this" >> icase( "that" );
```

In this regular expression, `"this"` will be matched exactly, but `"that"` will be matched irrespective of case.

Case-insensitive regular expressions raise the issue of internationalization: how should case-insensitive character comparisons be evaluated? Also, many character classes are locale-specific. Which characters are matched by `digit` and which are matched by `alpha`? The answer depends on the `std::locale` object the regular expression object is using. By default, all regular expression objects use the global locale. You can override the default by using the `imbue()` pattern modifier, as follows:

```
std::locale my_locale = /* initialize a std::locale object */;
sregex re = imbue( my_locale )( +alpha >> +digit );
```

This regular expression will evaluate `alpha` and `digit` according to `my_locale`. See the section on Localization and Regex Traits for more information about how to customize the behavior of your regexes.

Static xpressive Syntax Cheat Sheet

The table below lists the familiar regex constructs and their equivalents in static xpressive.

Table 4. Perl syntax vs. Static xpressive syntax

Perl	Static xpressive	Meaning
`.`	`_`	any character (assuming Perl's /s modifier).
`ab`	`a >> b`	sequencing of `a` and `b` sub-expressions.
`a|b`	`a | b`	alternation of `a` and `b` sub-expressions.
`(a)`	`(s1= a)`	group and capture a back-reference.
`(?:a)`	`(a)`	group and do not capture a back-reference.
`\1`	`s1`	a previously captured back-reference.
`a*`	`*a`	zero or more times, greedy.
`a+`	`+a`	one or more times, greedy.
`a?`	`!a`	zero or one time, greedy.
`a{n,m}`	`repeat<n,m>(a)`	between `n` and `m` times, greedy.
`a*?`	`-*a`	zero or more times, non-greedy.
`a+?`	`-+a`	one or more times, non-greedy.
`a??`	`-!a`	zero or one time, non-greedy.
`a{n,m}?`	`-repeat<n,m>(a)`	between `n` and `m` times, non-greedy.
`^`	`bos`	beginning of sequence assertion.
`$`	`eos`	end of sequence assertion.
`\b`	`_b`	word boundary assertion.
`\B`	`~_b`	not word boundary assertion.
`\n`	`_n`	literal newline.
`.`	`~_n`	any character except a literal newline (without Perl's /s modifier).
`\r?\n|\r`	`_ln`	logical newline.
`[^\r\n]`	`~_ln`	any single character not a logical newline.
`\w`	`_w`	a word character, equivalent to set[alnum | '_'].
`\W`	`~_w`	not a word character, equivalent to ~set[alnum | '_'].
`\d`	`_d`	a digit character.

Perl	Static xpressive	Meaning
`\D`	`~_d`	not a digit character.
`\s`	`_s`	a space character.
`\S`	`~_s`	not a space character.
`[:alnum:]`	`alnum`	an alpha-numeric character.
`[:alpha:]`	`alpha`	an alphabetic character.
`[:blank:]`	`blank`	a horizontal white-space character.
`[:cntrl:]`	`cntrl`	a control character.
`[:digit:]`	`digit`	a digit character.
`[:graph:]`	`graph`	a graphable character.
`[:lower:]`	`lower`	a lower-case character.
`[:print:]`	`print`	a printing character.
`[:punct:]`	`punct`	a punctuation character.
`[:space:]`	`space`	a white-space character.
`[:upper:]`	`upper`	an upper-case character.
`[:xdigit:]`	`xdigit`	a hexadecimal digit character.
`[0-9]`	`range('0','9')`	characters in range `'0'` through `'9'`.
`[abc]`	`as_xpr('a') | 'b' |'c'`	characters `'a'`, `'b'`, or `'c'`.
`[abc]`	`(set= 'a','b','c')`	*same as above*
`[0-9abc]`	`set[ range('0','9') | 'a' | 'b' | 'c' ]`	characters `'a'`, `'b'`, `'c'` or in range `'0'` through `'9'`.
`[0-9abc]`	`set[ range('0','9') | (set= 'a','b','c') ]`	*same as above*
`[^abc]`	`~(set= 'a','b','c')`	not characters `'a'`, `'b'`, or `'c'`.
`(?i:`*stuff*`)`	`icase(`*stuff*`)`	match *stuff* disregarding case.
`(?>`*stuff*`)`	`keep(`*stuff*`)`	independent sub-expression, match *stuff* and turn off backtracking.
`(?=`*stuff*`)`	`before(`*stuff*`)`	positive look-ahead assertion, match if before *stuff* but don't include *stuff* in the match.
`(?!`*stuff*`)`	`~before(`*stuff*`)`	negative look-ahead assertion, match if not before *stuff*.

Perl	Static xpressive	Meaning
`(?<=stuff)`	`after(stuff)`	positive look-behind assertion, match if after *stuff* but don't include *stuff* in the match. (*stuff* must be constant-width.)
`(?<!stuff)`	`~after(stuff)`	negative look-behind assertion, match if not after *stuff*. (*stuff* must be constant-width.)

`(?P<name>stuff)`	`mark_tag name(n);`
	`...`
	`(name= stuff)`
		Create a named capture.

`(?P=name)`	`mark_tag name(n);`
	`...`
	`name`
		Refer back to a previously created named capture.

Dynamic Regexes

Overview

Static regexes are dandy, but sometimes you need something a bit more ... dynamic. Imagine you are developing a text editor with a regex search/replace feature. You need to accept a regular expression from the end user as input at run-time. There should be a way to parse a string into a regular expression. That's what xpressive's dynamic regexes are for. They are built from the same core components as their static counterparts, but they are late-bound so you can specify them at run-time.

Construction and Assignment

There are two ways to create a dynamic regex: with the `basic_regex<>::compile()` function or with the `regex_compiler<>` class template. Use `basic_regex<>::compile()` if you want the default locale. Use `regex_compiler<>` if you need to specify a different locale. In the section on regex grammars, we'll see another use for `regex_compiler<>`.

Here is an example of using `basic_regex<>::compile()`:

```
sregex re = sregex::compile( "this|that", regex_constants::icase );
```

Here is the same example using `regex_compiler<>`:

```
sregex_compiler compiler;
sregex re = compiler.compile( "this|that", regex_constants::icase );
```

`basic_regex<>::compile()` is implemented in terms of `regex_compiler<>`.

Dynamic xpressive Syntax

Since the dynamic syntax is not constrained by the rules for valid C++ expressions, we are free to use familiar syntax for dynamic regexes. For this reason, the syntax used by xpressive for dynamic regexes follows the lead set by John Maddock's proposal to add regular expressions to the Standard Library. It is essentially the syntax standardized by ECMAScript, with minor changes in support of internationalization.

Since the syntax is documented exhaustively elsewhere, I will simply refer you to the existing standards, rather than duplicate the specification here.

Internationalization

As with static regexes, dynamic regexes support internationalization by allowing you to specify a different `std::locale`. To do this, you must use `regex_compiler<>`. The `regex_compiler<>` class has an `imbue()` function. After you have imbued a `regex_compiler<>` object with a custom `std::locale`, all regex objects compiled by that `regex_compiler<>` will use that locale. For example:

```
std::locale my_locale = /* initialize your locale object here */;
sregex_compiler compiler;
compiler.imbue( my_locale );
sregex re = compiler.compile( "\\w+|\\d+" );
```

This regex will use `my_locale` when evaluating the intrinsic character sets `"\\w"` and `"\\d"`.

Matching and Searching

Overview

Once you have created a regex object, you can use the `regex_match()` and `regex_search()` algorithms to find patterns in strings. This page covers the basics of regex matching and searching. In all cases, if you are familiar with how `regex_match()` and `regex_search()` in the Boost.Regex library work, xpressive's versions work the same way.

Seeing if a String Matches a Regex

The `regex_match()` algorithm checks to see if a regex matches a given input.

Warning

The `regex_match()` algorithm will only report success if the regex matches the *whole input*, from beginning to end. If the regex matches only a part of the input, `regex_match()` will return false. If you want to search through the string looking for sub-strings that the regex matches, use the `regex_search()` algorithm.

The input can be a bidirectional range such as `std::string`, a C-style null-terminated string or a pair of iterators. In all cases, the type of the iterator used to traverse the input sequence must match the iterator type used to declare the regex object. (You can use the table in the Quick Start to find the correct regex type for your iterator.)

```
cregex cre = +_w;  // this regex can match C-style strings
sregex sre = +_w;  // this regex can match std::strings

if( regex_match( "hello", cre ) )             // OK
    { /*...*/ }

if( regex_match( std::string("hello"), sre ) ) // OK
    { /*...*/ }

if( regex_match( "hello", sre ) )             // ERROR! iterator mis-match!
    { /*...*/ }
```

The `regex_match()` algorithm optionally accepts a `match_results<>` struct as an out parameter. If given, the `regex_match()` algorithm fills in the `match_results<>` struct with information about which parts of the regex matched which parts of the input.

```
cmatch what;
cregex cre = +(s1= _w);

// store the results of the regex_match in "what"
if( regex_match( "hello", what, cre ) )
{
    std::cout << what[1] << '\n'; // prints "o"
}
```

The `regex_match()` algorithm also optionally accepts a `match_flag_type` bitmask. With `match_flag_type`, you can control certain aspects of how the match is evaluated. See the `match_flag_type` reference for a complete list of the flags and their meanings.

```
std::string str("hello");
sregex sre = bol >> +_w;

// match_not_bol means that "bol" should not match at [begin,begin)
if( regex_match( str.begin(), str.end(), sre, regex_constants::match_not_bol ) )
{
    // should never get here!!!
}
```

Click here to see a complete example program that shows how to use `regex_match()`. And check the `regex_match()` reference to see a complete list of the available overloads.

Searching for Matching Sub-Strings

Use `regex_search()` when you want to know if an input sequence contains a sub-sequence that a regex matches. `regex_search()` will try to match the regex at the beginning of the input sequence and scan forward in the sequence until it either finds a match or exhausts the sequence.

In all other regards, `regex_search()` behaves like `regex_match()` *(see above)*. In particular, it can operate on a bidirectional range such as `std::string`, C-style null-terminated strings or iterator ranges. The same care must be taken to ensure that the iterator type of your regex matches the iterator type of your input sequence. As with `regex_match()`, you can optionally provide a `match_results<>` struct to receive the results of the search, and a `match_flag_type` bitmask to control how the match is evaluated.

Click here to see a complete example program that shows how to use `regex_search()`. And check the `regex_search()` reference to see a complete list of the available overloads.

Accessing Results

Overview

Sometimes, it is not enough to know simply whether a `regex_match()` or `regex_search()` was successful or not. If you pass an object of type `match_results<>` to `regex_match()` or `regex_search()`, then after the algorithm has completed successfully the `match_results<>` will contain extra information about which parts of the regex matched which parts of the sequence. In Perl, these sub-sequences are called *back-references*, and they are stored in the variables `$1`, `$2`, etc. In xpressive, they are objects of type `sub_match<>`, and they are stored in the `match_results<>` structure, which acts as a vector of `sub_match<>` objects.

match_results

So, you've passed a `match_results<>` object to a regex algorithm, and the algorithm has succeeded. Now you want to examine the results. Most of what you'll be doing with the `match_results<>` object is indexing into it to access its internally stored `sub_match<>` objects, but there are a few other things you can do with a `match_results<>` object besides.

The table below shows how to access the information stored in a `match_results<>` object named `what`.

Table 5. match_results<> Accessors

Accessor	Effects
`what.size()`	Returns the number of sub-matches, which is always greater than zero after a successful match because the full match is stored in the zero-th sub-match.
`what[n]`	Returns the *n*-th sub-match.
`what.length(n)`	Returns the length of the *n*-th sub-match. Same as `what[n].length()`.
`what.position(n)`	Returns the offset into the input sequence at which the *n*-th sub-match begins.
`what.str(n)`	Returns a `std::basic_string<>` constructed from the *n*-th sub-match. Same as `what[n].str()`.
`what.prefix()`	Returns a `sub_match<>` object which represents the sub-sequence from the beginning of the input sequence to the start of the full match.
`what.suffix()`	Returns a `sub_match<>` object which represents the sub-sequence from the end of the full match to the end of the input sequence.
`what.regex_id()`	Returns the `regex_id` of the `basic_regex<>` object that was last used with this `match_results<>` object.

There is more you can do with the `match_results<>` object, but that will be covered when we talk about Grammars and Nested Matches.

sub_match

When you index into a `match_results<>` object, you get back a `sub_match<>` object. A `sub_match<>` is basically a pair of iterators. It is defined like this:

```
template< class BidirectionalIterator >
struct sub_match
    : std::pair< BidirectionalIterator, BidirectionalIterator >
{
    bool matched;
    // ...
};
```

Since it inherits publicaly from `std::pair<>`, `sub_match<>` has `first` and `second` data members of type `BidirectionalIterator`. These are the beginning and end of the sub-sequence this `sub_match<>` represents. `sub_match<>` also has a Boolean `matched` data member, which is true if this `sub_match<>` participated in the full match.

The following table shows how you might access the information stored in a `sub_match<>` object called `sub`.

Table 6. sub_match<> Accessors

Accessor	Effects
`sub.length()`	Returns the length of the sub-match. Same as `std::distance(sub.first,sub.second)`.
`sub.str()`	Returns a `std::basic_string<>` constructed from the sub-match. Same as `std::basic_string<char_type>(sub.first,sub.second)`.
`sub.compare(str)`	Performs a string comparison between the sub-match and `str`, where `str` can be a `std::basic_string<>`, C-style null-terminated string, or another sub-match. Same as `sub.str().compare(str)`.

Results Invalidation

Results are stored as iterators into the input sequence. Anything which invalidates the input sequence will invalidate the match results. For instance, if you match a `std::string` object, the results are only valid until your next call to a non-const member function of that `std::string` object. After that, the results held by the `match_results<>` object are invalid. Don't use them!

String Substitutions

Regular expressions are not only good for searching text; they're good at *manipulating* it. And one of the most common text manipulation tasks is search-and-replace. xpressive provides the `regex_replace()` algorithm for searching and replacing.

regex_replace()

Performing search-and-replace using `regex_replace()` is simple. All you need is an input sequence, a regex object, and a format string or a formatter object. There are several versions of the `regex_replace()` algorithm. Some accept the input sequence as a bidirectional container such as `std::string` and returns the result in a new container of the same type. Others accept the input as a null terminated string and return a `std::string`. Still others accept the input sequence as a pair of iterators and writes the result into an output iterator. The substitution may be specified as a string with format sequences or as a formatter object. Below are some simple examples of using string-based substitutions.

```
std::string input("This is his face");
sregex re = as_xpr("his");               // find all occurrences of "his" ...
std::string format("her");               // ... and replace them with "her"

// use the version of regex_replace() that operates on strings
std::string output = regex_replace( input, re, format );
std::cout << output << '\n';

// use the version of regex_replace() that operates on iterators
std::ostream_iterator< char > out_iter( std::cout );
regex_replace( out_iter, input.begin(), input.end(), re, format );
```

The above program prints out the following:

```
Ther is her face
Ther is her face
```

Notice that *all* the occurrences of `"his"` have been replaced with `"her"`.

Click here to see a complete example program that shows how to use `regex_replace()`. And check the `regex_replace()` reference to see a complete list of the available overloads.

Replace Options

The `regex_replace()` algorithm takes an optional bitmask parameter to control the formatting. The possible values of the bitmask are:

Table 7. Format Flags

Flag	Meaning
`format_default`	Recognize the ECMA-262 format sequences (see below).
`format_first_only`	Only replace the first match, not all of them.
`format_no_copy`	Don't copy the parts of the input sequence that didn't match the regex to the output sequence.
`format_literal`	Treat the format string as a literal; that is, don't recognize any escape sequences.
`format_perl`	Recognize the Perl format sequences (see below).
`format_sed`	Recognize the sed format sequences (see below).
`format_all`	In addition to the Perl format sequences, recognize some Boost-specific format sequences.

These flags live in the `xpressive::regex_constants` namespace. If the substitution parameter is a function object instead of a string, the flags `format_literal`, `format_perl`, `format_sed`, and `format_all` are ignored.

The ECMA-262 Format Sequences

When you haven't specified a substitution string dialect with one of the format flags above, you get the dialect defined by ECMA-262, the standard for ECMAScript. The table below shows the escape sequences recognized in ECMA-262 mode.

Table 8. Format Escape Sequences

Escape Sequence	Meaning
`$1`, `$2`, etc.	the corresponding sub-match
`$&`	the full match
`$``	the match prefix
`$'`	the match suffix
`$$`	a literal `'$'` character

Any other sequence beginning with `'$'` simply represents itself. For example, if the format string were `"$a"` then `"$a"` would be inserted into the output sequence.

The Sed Format Sequences

When specifying the `format_sed` flag to `regex_replace()`, the following escape sequences are recognized:

Table 9. Sed Format Escape Sequences

Escape Sequence	Meaning
`\1`, `\2`, etc.	The corresponding sub-match
`&`	the full match
`\a`	A literal `'\a'`
`\e`	A literal `char_type(27)`
`\f`	A literal `'\f'`
`\n`	A literal `'\n'`
`\r`	A literal `'\r'`
`\t`	A literal `'\t'`
`\v`	A literal `'\v'`
`\xFF`	A literal `char_type(0xFF)`, where *F* is any hex digit
`\x{FFFF}`	A literal `char_type(0xFFFF)`, where *F* is any hex digit
`\cX`	The control character *X*

The Perl Format Sequences

When specifying the `format_perl` flag to `regex_replace()`, the following escape sequences are recognized:

Table 10. Perl Format Escape Sequences

Escape Sequence	Meaning
`$1`, `$2`, etc.	the corresponding sub-match
`$&`	the full match
`$\``	the match prefix
`$'`	the match suffix
`$$`	a literal `'$'` character
`\a`	A literal `'\a'`
`\e`	A literal `char_type(27)`
`\f`	A literal `'\f'`
`\n`	A literal `'\n'`
`\r`	A literal `'\r'`
`\t`	A literal `'\t'`
`\v`	A literal `'\v'`
`\xFF`	A literal `char_type(0xFF)`, where *F* is any hex digit
`\x{FFFF}`	A literal `char_type(0xFFFF)`, where *F* is any hex digit
`\cX`	The control character *X*
`\l`	Make the next character lowercase
`\L`	Make the rest of the substitution lowercase until the next `\E`
`\u`	Make the next character uppercase
`\U`	Make the rest of the substitution uppercase until the next `\E`
`\E`	Terminate `\L` or `\U`
`\1`, `\2`, etc.	The corresponding sub-match
`\g<name>`	The named backref *name*

The Boost-Specific Format Sequences

When specifying the `format_all` flag to `regex_replace()`, the escape sequences recognized are the same as those above for `format_perl`. In addition, conditional expressions of the following form are recognized:

```
?Ntrue-expression:false-expression
```

where *N* is a decimal digit representing a sub-match. If the corresponding sub-match participated in the full match, then the substitution is *true-expression*. Otherwise, it is *false-expression*. In this mode, you can use parens `()` for grouping. If you want a literal paren, you must escape it as `\(`.

Formatter Objects

Format strings are not always expressive enough for all your text substitution needs. Consider the simple example of wanting to map input strings to output strings, as you may want to do with environment variables. Rather than a format *string*, for this you would use a formatter *object*. Consider the following code, which finds embedded environment variables of the form `"$(XYZ)"` and computes the substitution string by looking up the environment variable in a map.

```
#include <map>
#include <string>
#include <iostream>
#include <boost/xpressive/xpressive.hpp>
using namespace boost;
using namespace xpressive;

std::map<std::string, std::string> env;

std::string const &format_fun(smatch const &what)
{
    return env[what[1].str()];
}

int main()
{
    env["X"] = "this";
    env["Y"] = "that";

    std::string input("\"$(X)\" has the value \"$(Y)\"");

    // replace strings like "$(XYZ)" with the result of env["XYZ"]
    sregex envar = "$(" >> (s1 = +_w) >> ')';
    std::string output = regex_replace(input, envar, format_fun);
    std::cout << output << std::endl;
}
```

In this case, we use a function, `format_fun()` to compute the substitution string on the fly. It accepts a `match_results<>` object which contains the results of the current match. `format_fun()` uses the first submatch as a key into the global `env` map. The above code displays:

```
"this" has the value "that"
```

The formatter need not be an ordinary function. It may be an object of class type. And rather than return a string, it may accept an output iterator into which it writes the substitution. Consider the following, which is functionally equivalent to the above.

```
#include <map>
#include <string>
#include <iostream>
#include <boost/xpressive/xpressive.hpp>
using namespace boost;
using namespace xpressive;

struct formatter
{
    typedef std::map<std::string, std::string> env_map;
    env_map env;

    template<typename Out>
    Out operator()(smatch const &what, Out out) const
    {
        env_map::const_iterator where = env.find(what[1]);
        if(where != env.end())
        {
            std::string const &sub = where->second;
            out = std::copy(sub.begin(), sub.end(), out);
        }
        return out;
    }

};

int main()
{
    formatter fmt;
    fmt.env["X"] = "this";
    fmt.env["Y"] = "that";

    std::string input("\"$(X)\" has the value \"$(Y)\"");

    sregex envar = "$(" >> (s1 = +_w) >> ')';
    std::string output = regex_replace(input, envar, fmt);
    std::cout << output << std::endl;
}
```

The formatter must be a callable object -- a function or a function object -- that has one of three possible signatures, detailed in the table below. For the table, `fmt` is a function pointer or function object, `what` is a `match_results<>` object, `out` is an OutputIterator, and `flags` is a value of `regex_constants::match_flag_type`:

Table 11. Formatter Signatures

Formatter Invocation	Return Type	Semantics
`fmt(what)`	Range of characters (e.g. `std::string`) or null-terminated string	The string matched by the regex is replaced with the string returned by the formatter.
`fmt(what, out)`	OutputIterator	The formatter writes the replacement string into `out` and returns `out`.
`fmt(what, out, flags)`	OutputIterator	The formatter writes the replacement string into `out` and returns `out`. The `flags` parameter is the value of the match flags passed to the `regex_replace()` algorithm.

Formatter Expressions

In addition to format *strings* and formatter *objects*, `regex_replace()` also accepts formatter *expressions*. A formatter expression is a lambda expression that generates a string. It uses the same syntax as that for Semantic Actions, which are covered later. The above example, which uses `regex_replace()` to substitute strings for environment variables, is repeated here using a formatter expression.

```
#include <map>
#include <string>
#include <iostream>
#include <boost/xpressive/xpressive.hpp>
#include <boost/xpressive/regex_actions.hpp>
using namespace boost::xpressive;

int main()
{
    std::map<std::string, std::string> env;
    env["X"] = "this";
    env["Y"] = "that";

    std::string input("\"$(X)\" has the value \"$(Y)\"");

    sregex envar = "$(" >> (s1 = +_w) >> ')';
    std::string output = regex_replace(input, envar, ref(env)[s1]);
    std::cout << output << std::endl;
}
```

In the above, the formatter expression is `ref(env)[s1]`. This means to use the value of the first submatch, `s1`, as a key into the `env` map. The purpose of `xpressive::ref()` here is to make the reference to the `env` local variable *lazy* so that the index operation is deferred until we know what to replace `s1` with.

String Splitting and Tokenization

`regex_token_iterator<>` is the Ginsu knife of the text manipulation world. It slices! It dices! This section describes how to use the highly-configurable `regex_token_iterator<>` to chop up input sequences.

Overview

You initialize a `regex_token_iterator<>` with an input sequence, a regex, and some optional configuration parameters. The `regex_token_iterator<>` will use `regex_search()` to find the first place in the sequence that the regex matches. When dereferenced, the `regex_token_iterator<>` returns a *token* in the form of a `std::basic_string<>`. Which string it returns depends on the configuration parameters. By default it returns a string corresponding to the full match, but it could also return a string corresponding to a particular marked sub-expression, or even the part of the sequence that *didn't* match. When you increment the `regex_token_iterator<>`, it will move to the next token. Which token is next depends on the configuration parameters. It could simply be a different marked sub-expression in the current match, or it could be part or all of the next match. Or it could be the part that *didn't* match.

As you can see, `regex_token_iterator<>` can do a lot. That makes it hard to describe, but some examples should make it clear.

Example 1: Simple Tokenization

This example uses `regex_token_iterator<>` to chop a sequence into a series of tokens consisting of words.

```
std::string input("This is his face");
sregex re = +_w;                      // find a word

// iterate over all the words in the input
sregex_token_iterator begin( input.begin(), input.end(), re ), end;

// write all the words to std::cout
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

This program displays the following:

```
This
is
his
face
```

Example 2: Simple Tokenization, Reloaded

This example also uses `regex_token_iterator<>` to chop a sequence into a series of tokens consisting of words, but it uses the regex as a delimiter. When we pass a `-1` as the last parameter to the `regex_token_iterator<>` constructor, it instructs the token iterator to consider as tokens those parts of the input that *didn't* match the regex.

```
std::string input("This is his face");
sregex re = +_s;                      // find white space

// iterate over all non-white space in the input. Note the -1 below:
sregex_token_iterator begin( input.begin(), input.end(), re, -1 ), end;

// write all the words to std::cout
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

This program displays the following:

```
This
is
his
face
```

Example 3: Simple Tokenization, Revolutions

This example also uses `regex_token_iterator<>` to chop a sequence containing a bunch of dates into a series of tokens consisting of just the years. When we pass a positive integer *N* as the last parameter to the `regex_token_iterator<>` constructor, it instructs the token iterator to consider as tokens only the *N*-th marked sub-expression of each match.

```
std::string input("01/02/2003 blahblah 04/23/1999 blahblah 11/13/1981");
sregex re = sregex::compile("(\\d{2})/(\\d{2})/(\\d{4})"); // find a date

// iterate over all the years in the input. Note the 3 below, corresponding to the 3rd sub-ex↵
pression:
sregex_token_iterator begin( input.begin(), input.end(), re, 3 ), end;

// write all the words to std::cout
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

This program displays the following:

```
2003
1999
1981
```

Example 4: Not-So-Simple Tokenization

This example is like the previous one, except that instead of tokenizing just the years, this program turns the days, months and years into tokens. When we pass an array of integers *{I,J,...}* as the last parameter to the `regex_token_iterator<>` constructor, it instructs the token iterator to consider as tokens the *I*-th, *J*-th, etc. marked sub-expression of each match.

```
std::string input("01/02/2003 blahblah 04/23/1999 blahblah 11/13/1981");
sregex re = sregex::compile("(\\d{2})/(\\d{2})/(\\d{4})"); // find a date

// iterate over the days, months and years in the input
int const sub_matches[] = { 2, 1, 3 }; // day, month, year
sregex_token_iterator begin( input.begin(), input.end(), re, sub_matches ), end;

// write all the words to std::cout
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

This program displays the following:

```
02
01
2003
23
04
1999
13
11
1981
```

The `sub_matches` array instructs the `regex_token_iterator<>` to first take the value of the 2nd sub-match, then the 1st sub-match, and finally the 3rd. Incrementing the iterator again instructs it to use `regex_search()` again to find the next match. At that point, the process repeats -- the token iterator takes the value of the 2nd sub-match, then the 1st, et cetera.

Named Captures

Overview

For complicated regular expressions, dealing with numbered captures can be a pain. Counting left parentheses to figure out which capture to reference is no fun. Less fun is the fact that merely editing a regular expression could cause a capture to be assigned a new number, invaliding code that refers back to it by the old number.

Other regular expression engines solve this problem with a feature called *named captures*. This feature allows you to assign a name to a capture, and to refer back to the capture by name rather by number. Xpressive also supports named captures, both in dynamic and in static regexes.

Dynamic Named Captures

For dynamic regular expressions, xpressive follows the lead of other popular regex engines with the syntax of named captures. You can create a named capture with `"(?P<xxx>...)"` and refer back to that capture with `"(?P=xxx)"`. Here, for instance, is a regular expression that creates a named capture and refers back to it:

```
// Create a named capture called "char" that matches a single
// character and refer back to that capture by name.
sregex rx = sregex::compile("(?P<char>.)(?P=char)");
```

The effect of the above regular expression is to find the first doubled character.

Once you have executed a match or search operation using a regex with named captures, you can access the named capture through the `match_results<>` object using the capture's name.

```
std::string str("tweet");
sregex rx = sregex::compile("(?P<char>.)(?P=char)");
smatch what;
if(regex_search(str, what, rx))
{
    std::cout << "char = " << what["char"] << std::endl;
}
```

The above code displays:

```
char = e
```

You can also refer back to a named capture from within a substitution string. The syntax for that is `"\\g<xxx>"`. Below is some code that demonstrates how to use named captures when doing string substitution.

```
std::string str("tweet");
sregex rx = sregex::compile("(?P<char>.)(?P=char)");
str = regex_replace(str, rx, "**\\g<char>**", regex_constants::format_perl);
std::cout << str << std::endl;
```

Notice that you have to specify `format_perl` when using named captures. Only the perl syntax recognizes the `"\\g<xxx>"` syntax. The above code displays:

```
tw**e**t
```

Static Named Captures

If you're using static regular expressions, creating and using named captures is even easier. You can use the `mark_tag` type to create a variable that you can use like `s1`, `s2` and friends, but with a name that is more meaningful. Below is how the above example would look using static regexes:

```
mark_tag char_(1); // char_ is now a synonym for s1
sregex rx = (char_= _) >> char_;
```

After a match operation, you can use the `mark_tag` to index into the `match_results<>` to access the named capture:

```
std::string str("tweet");
mark_tag char_(1);
sregex rx = (char_= _) >> char_;
smatch what;
if(regex_search(str, what, rx))
{
    std::cout << what[char_] << std::endl;
}
```

The above code displays:

```
char = e
```

When doing string substitutions with `regex_replace()`, you can use named captures to create *format expressions* as below:

```
std::string str("tweet");
mark_tag char_(1);
sregex rx = (char_= _) >> char_;
str = regex_replace(str, rx, "**" + char_ + "**");
std::cout << str << std::endl;
```

The above code displays:

```
tw**e**t
```

Note

You need to include `<boost/xpressive/regex_actions.hpp>` to use format expressions.

Grammars and Nested Matches

Overview

One of the key benefits of representing regexes as C++ expressions is the ability to easily refer to other C++ code and data from within the regex. This enables programming idioms that are not possible with other regular expression libraries. Of particular note is the ability for one regex to refer to another regex, allowing you to build grammars out of regular expressions. This section describes how to embed one regex in another by value and by reference, how regex objects behave when they refer to other regexes, and how to access the tree of results after a successful parse.

Embedding a Regex by Value

The `basic_regex<>` object has value semantics. When a regex object appears on the right-hand side in the definition of another regex, it is as if the regex were embedded by value; that is, a copy of the nested regex is stored by the enclosing regex. The inner regex is invoked by the outer regex during pattern matching. The inner regex participates fully in the match, back-tracking as needed to make the match succeed.

Consider a text editor that has a regex-find feature with a whole-word option. You can implement this with xpressive as follows:

```
find_dialog dlg;
if( dialog_ok == dlg.do_modal() )
{
    std::string pattern = dlg.get_text();          // the pattern the user entered
    bool whole_word = dlg.whole_word.is_checked(); // did the user select the whole-word option?

    sregex re = sregex::compile( pattern );        // try to compile the pattern

    if( whole_word )
    {
        // wrap the regex in begin-word / end-word assertions
        re = bow >> re >> eow;
    }

    // ... use re ...
}
```

Look closely at this line:

```
// wrap the regex in begin-word / end-word assertions
re = bow >> re >> eow;
```

This line creates a new regex that embeds the old regex by value. Then, the new regex is assigned back to the original regex. Since a copy of the old regex was made on the right-hand side, this works as you might expect: the new regex has the behavior of the old regex wrapped in begin- and end-word assertions.

Note

Note that `re = bow >> re >> eow` does *not* define a recursive regular expression, since regex objects embed by value by default. The next section shows how to define a recursive regular expression by embedding a regex by reference.

Embedding a Regex by Reference

If you want to be able to build recursive regular expressions and context-free grammars, embedding a regex by value is not enough. You need to be able to make your regular expressions self-referential. Most regular expression engines don't give you that power, but xpressive does.

Tip

The theoretical computer scientists out there will correctly point out that a self-referential regular expression is not "regular", so in the strict sense, xpressive isn't really a *regular* expression engine at all. But as Larry Wall once said, "the term [regular expression] has grown with the capabilities of our pattern matching engines, so I'm not going to try to fight linguistic necessity here."

Consider the following code, which uses the `by_ref()` helper to define a recursive regular expression that matches balanced, nested parentheses:

```
sregex parentheses;
parentheses                          // A balanced set of parentheses ...
    = '('                            // is an opening parenthesis ...
       >>                            // followed by ...
        *(                           // zero or more ...
           keep( +~(set='(',')') ) ) // of a bunch of things that are not parentheses ...
         |                           // or ...
           by_ref(parentheses)       // a balanced set of parentheses
         )                           //   (ooh, recursion!) ...
       >>                            // followed by ...
      ')'                            // a closing parenthesis
    ;
```

Matching balanced, nested tags is an important text processing task, and it is one that "classic" regular expressions cannot do. The `by_ref()` helper makes it possible. It allows one regex object to be embedded in another *by reference*. Since the right-hand side holds `parentheses` by reference, assigning the right-hand side back to `parentheses` creates a cycle, which will execute recursively.

Building a Grammar

Once we allow self-reference in our regular expressions, the genie is out of the bottle and all manner of fun things are possible. In particular, we can now build grammars out of regular expressions. Let's have a look at the text-book grammar example: the humble calculator.

```
sregex group, factor, term, expression;

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

The regex `expression` defined above does something rather remarkable for a regular expression: it matches mathematical expressions. For example, if the input string were `"foo 9*(10+3) bar"`, this pattern would match `"9*(10+3)"`. It only matches well-formed mathematical expressions, where the parentheses are balanced and the infix operators have two arguments each. Don't try this with just any regular expression engine!

Let's take a closer look at this regular expression grammar. Notice that it is cyclic: `expression` is implemented in terms of `term`, which is implemented in terms of `factor`, which is implemented in terms of `group`, which is implemented in terms of `expression`, closing the loop. In general, the way to define a cyclic grammar is to forward-declare the regex objects and embed by reference those regular expressions that have not yet been initialized. In the above grammar, there is only one place where we need to reference a regex object that has not yet been initialized: the definition of `group`. In that place, we use `by_ref()` to embed `expression` by reference. In all other places, it is sufficient to embed the other regex objects by value, since they have already been initialized and their values will not change.

Tip

Embed by value if possible
In general, prefer embedding regular expressions by value rather than by reference. It involves one less indirection, making your patterns match a little faster. Besides, value semantics are simpler and will make your grammars easier to reason about. Don't worry about the expense of "copying" a regex. Each regex object shares its implementation with all of its copies.

Dynamic Regex Grammars

Using `regex_compiler<>`, you can also build grammars out of dynamic regular expressions. You do that by creating named regexes, and referring to other regexes by name. Each `regex_compiler<>` instance keeps a mapping from names to regexes that have been created with it.

You can create a named dynamic regex by prefacing your regex with `"(?$name=)"`, where *name* is the name of the regex. You can refer to a named regex from another regex with `"(?$name)"`. The named regex does not need to exist yet at the time it is referenced in another regex, but it must exist by the time you use the regex.

Below is a code fragment that uses dynamic regex grammars to implement the calculator example from above.

```
using namespace boost::xpressive;
using namespace regex_constants;

sregex expr;

{
     sregex_compiler compiler;
     syntax_option_type x = ignore_white_space;

            compiler.compile("(? $group  = ) \\( (? $expr ) \\) ", x);
            compiler.compile("(? $factor = ) \\d+ | (? $group ) ", x);
            compiler.compile("(? $term   = ) (? $factor )"
                             " ( \\* (? $factor ) | / (? $factor ) )* ", x);
     expr = compiler.compile("(? $expr   = ) (? $term )"
                             "   ( \\+ (? $term ) | - (? $term )   )* ", x);
}

std::string str("foo 9*(10+3) bar");
smatch what;

if(regex_search(str, what, expr))
{
     // This prints "9*(10+3)":
     std::cout << what[0] << std::endl;
}
```

As with static regex grammars, nested regex invocations create nested match results (see *Nested Results* below). The result is a complete parse tree for string that matched. Unlike static regexes, dynamic regexes are always embedded by reference, not by value.

Cyclic Patterns, Copying and Memory Management, Oh My!

The calculator examples above raises a number of very complicated memory-management issues. Each of the four regex objects refer to each other, some directly and some indirectly, some by value and some by reference. What if we were to return one of them from a function and let the others go out of scope? What becomes of the references? The answer is that the regex objects are internally reference counted, such that they keep their referenced regex objects alive as long as they need them. So passing a regex object by value is never a problem, even if it refers to other regex objects that have gone out of scope.

Those of you who have dealt with reference counting are probably familiar with its Achilles Heel: cyclic references. If regex objects are reference counted, what happens to cycles like the one created in the calculator examples? Are they leaked? The answer is no, they are not leaked. The `basic_regex<>` object has some tricky reference tracking code that ensures that even cyclic regex grammars are cleaned up when the last external reference goes away. So don't worry about it. Create cyclic grammars, pass your regex objects around and copy them all you want. It is fast and efficient and guaranteed not to leak or result in dangling references.

Nested Regexes and Sub-Match Scoping

Nested regular expressions raise the issue of sub-match scoping. If both the inner and outer regex write to and read from the same sub-match vector, chaos would ensue. The inner regex would stomp on the sub-matches written by the outer regex. For example, what does this do?

```
sregex inner = sregex::compile( "(.)\\1" );
sregex outer = (s1= _) >> inner >> s1;
```

The author probably didn't intend for the inner regex to overwrite the sub-match written by the outer regex. The problem is particularly acute when the inner regex is accepted from the user as input. The author has no way of knowing whether the inner regex will stomp the sub-match vector or not. This is clearly not acceptable.

Instead, what actually happens is that each invocation of a nested regex gets its own scope. Sub-matches belong to that scope. That is, each nested regex invocation gets its own copy of the sub-match vector to play with, so there is no way for an inner regex to stomp on the sub-matches of an outer regex. So, for example, the regex `outer` defined above would match `"ABBA"`, as it should.

Nested Results

If nested regexes have their own sub-matches, there should be a way to access them after a successful match. In fact, there is. After a `regex_match()` or `regex_search()`, the `match_results<>` struct behaves like the head of a tree of nested results. The `match_results<>` class provides a `nested_results()` member function that returns an ordered sequence of `match_results<>` structures, representing the results of the nested regexes. The order of the nested results is the same as the order in which the nested regex objects matched.

Take as an example the regex for balanced, nested parentheses we saw earlier:

```
sregex parentheses;
parentheses = '(' >> *( keep( +~(set='(',')') ) | by_ref(parentheses) ) >> ')';

smatch what;
std::string str( "blah blah( a(b)c (c(e)f (g)h )i (j)6 )blah" );

if( regex_search( str, what, parentheses ) )
{
    // display the whole match
    std::cout << what[0] << '\n';

    // display the nested results
    std::for_each(
        what.nested_results().begin(),
        what.nested_results().end(),
        output_nested_results() );
}
```

This program displays the following:

```
( a(b)c (c(e)f (g)h )i (j)6 )
    (b)
    (c(e)f (g)h )
        (e)
        (g)
    (j)
```

Here you can see how the results are nested and that they are stored in the order in which they are found.

Tip

See the definition of output_nested_results in the Examples section.

Filtering Nested Results

Sometimes a regex will have several nested regex objects, and you want to know which result corresponds to which regex object. That's where `basic_regex<>::regex_id()` and `match_results<>::regex_id()` come in handy. When iterating over the nested results, you can compare the regex id from the results to the id of the regex object you're interested in.

To make this a bit easier, xpressive provides a predicate to make it simple to iterate over just the results that correspond to a certain nested regex. It is called `regex_id_filter_predicate`, and it is intended to be used with Boost.Iterator. You can use it as follows:

```
sregex name = +alpha;
sregex integer = +_d;
sregex re = *( *_s >> ( name | integer ) );

smatch what;
std::string str( "marsha 123 jan 456 cindy 789" );

if( regex_match( str, what, re ) )
{
    smatch::nested_results_type::const_iterator begin = what.nested_results().begin();
    smatch::nested_results_type::const_iterator end   = what.nested_results().end();

    // declare filter predicates to select just the names or the integers
    sregex_id_filter_predicate name_id( name.regex_id() );
    sregex_id_filter_predicate integer_id( integer.regex_id() );

    // iterate over only the results from the name regex
    std::for_each(
        boost::make_filter_iterator( name_id, begin, end ),
        boost::make_filter_iterator( name_id, end, end ),
        output_result
        );

    std::cout << '\n';

    // iterate over only the results from the integer regex
    std::for_each(
        boost::make_filter_iterator( integer_id, begin, end ),
        boost::make_filter_iterator( integer_id, end, end ),
        output_result
        );
}
```

where `output_results` is a simple function that takes a `smatch` and displays the full match. Notice how we use the `regex_id_filter_predicate` together with `basic_regex<>::regex_id()` and `boost::make_filter_iterator()` from the Boost.Iterator to select only those results corresponding to a particular nested regex. This program displays the following:

```
marsha
jan
cindy
123
456
789
```

Semantic Actions and User-Defined Assertions

Overview

Imagine you want to parse an input string and build a `std::map<>` from it. For something like that, matching a regular expression isn't enough. You want to *do something* when parts of your regular expression match. Xpressive lets you attach semantic actions to parts of your static regular expressions. This section shows you how.

Semantic Actions

Consider the following code, which uses xpressive's semantic actions to parse a string of word/integer pairs and stuffs them into a `std::map<>`. It is described below.

```
#include <string>
#include <iostream>
#include <boost/xpressive/xpressive.hpp>
#include <boost/xpressive/regex_actions.hpp>
using namespace boost::xpressive;

int main()
{
    std::map<std::string, int> result;
    std::string str("aaa=>1 bbb=>23 ccc=>456");

    // Match a word and an integer, separated by =>,
    // and then stuff the result into a std::map<>
    sregex pair = ( (s1= +_w) >> "=>" >> (s2= +_d) )
        [ ref(result)[s1] = as<int>(s2) ];

    // Match one or more word/integer pairs, separated
    // by whitespace.
    sregex rx = pair >> *(+_s >> pair);

    if(regex_match(str, rx))
    {
        std::cout << result["aaa"] << '\n';
        std::cout << result["bbb"] << '\n';
        std::cout << result["ccc"] << '\n';
    }

    return 0;
}
```

This program prints the following:

```
1
23
456
```

The regular expression `pair` has two parts: the pattern and the action. The pattern says to match a word, capturing it in sub-match 1, and an integer, capturing it in sub-match 2, separated by `"=>"`. The action is the part in square brackets: `[ ref(result)[s1] = as<int>(s2) ]`. It says to take sub-match one and use it to index into the `results` map, and assign to it the result of converting sub-match 2 to an integer.

Note

To use semantic actions with your static regexes, you must `#include <boost/xpressive/regex_actions.hpp>`

How does this work? Just as the rest of the static regular expression, the part between brackets is an expression template. It encodes the action and executes it later. The expression `ref(result)` creates a lazy reference to the `result` object. The larger expression `ref(result)[s1]` is a lazy map index operation. Later, when this action is getting executed, `s1` gets replaced with the first `sub_match<>`. Likewise, when `as<int>(s2)` gets executed, `s2` is replaced with the second `sub_match<>`. The `as<>` action converts its argument to the requested type using Boost.Lexical_cast. The effect of the whole action is to insert a new word/integer pair into the map.

Note

There is an important difference between the function `boost::ref()` in `<boost/ref.hpp>` and `boost::xpressive::ref()` in `<boost/xpressive/regex_actions.hpp>`. The first returns a plain `reference_wrapper<>` which behaves in many respects like an ordinary reference. By contrast, `boost::xpressive::ref()` returns a *lazy* reference that you can use in expressions that are executed lazily. That is why we can say `ref(result)[s1]`, even though `result` doesn't have an `operator[]` that would accept `s1`.

In addition to the sub-match placeholders `s1`, `s2`, etc., you can also use the placeholder _ within an action to refer back to the string matched by the sub-expression to which the action is attached. For instance, you can use the following regex to match a bunch of digits, interpret them as an integer and assign the result to a local variable:

```
int i = 0;
// Here, _ refers back to all the
// characters matched by (+_d)
sregex rex = (+_d)[ ref(i) = as<int>(_) ];
```

Lazy Action Execution

What does it mean, exactly, to attach an action to part of a regular expression and perform a match? When does the action execute? If the action is part of a repeated sub-expression, does the action execute once or many times? And if the sub-expression initially matches, but ultimately fails because the rest of the regular expression fails to match, is the action executed at all?

The answer is that by default, actions are executed *lazily*. When a sub-expression matches a string, its action is placed on a queue, along with the current values of any sub-matches to which the action refers. If the match algorithm must backtrack, actions are popped off the queue as necessary. Only after the entire regex has matched successfully are the actions actually exeucted. They are executed all at once, in the order in which they were added to the queue, as the last step before `regex_match()` returns.

For example, consider the following regex that increments a counter whenever it finds a digit.

```
int i = 0;
std::string str("1!2!3?");
// count the exciting digits, but not the
// questionable ones.
sregex rex = +( _d [ ++ref(i) ] >> '!' );
regex_search(str, rex);
assert( i == 2 );
```

The action `++ref(i)` is queued three times: once for each found digit. But it is only *executed* twice: once for each digit that precedes a `'!'` character. When the `'?'` character is encountered, the match algorithm backtracks, removing the final action from the queue.

Immediate Action Execution

When you want semantic actions to execute immediately, you can wrap the sub-expression containing the action in a `keep()`. `keep()` turns off back-tracking for its sub-expression, but it also causes any actions queued by the sub-expression to execute at the end of the `keep()`. It is as if the sub-expression in the `keep()` were compiled into an independent regex object, and matching the `keep()` is like a separate invocation of `regex_search()`. It matches characters and executes actions but never backtracks or unwinds. For example, imagine the above example had been written as follows:

```
int i = 0;
std::string str("1!2!3?");
// count all the digits.
sregex rex = +( keep( _d [ ++ref(i) ] ) >> '!' );
regex_search(str, rex);
assert( i == 3 );
```

We have wrapped the sub-expression `_d [ ++ref(i) ]` in `keep()`. Now, whenever this regex matches a digit, the action will be queued and then immediately executed before we try to match a `'!'` character. In this case, the action executes three times.

Note

Like `keep()`, actions within `before()` and `after()` are also executed early when their sub-expressions have matched.

Lazy Functions

So far, we've seen how to write semantic actions consisting of variables and operators. But what if you want to be able to call a function from a semantic action? Xpressive provides a mechanism to do this.

The first step is to define a function object type. Here, for instance, is a function object type that calls `push()` on its argument:

```
struct push_impl
{
    // Result type, needed for tr1::result_of
    typedef void result_type;

    template<typename Sequence, typename Value>
    void operator()(Sequence &seq, Value const &val) const
    {
        seq.push(val);
    }
};
```

The next step is to use xpressive's `function<>` template to define a function object named `push`:

```
// Global "push" function object.
function<push_impl>::type const push = {{}};
```

The initialization looks a bit odd, but this is because `push` is being statically initialized. That means it doesn't need to be constructed at runtime. We can use `push` in semantic actions as follows:

```
std::stack<int> ints;
// Match digits, cast them to an int
// and push it on the stack.
sregex rex = (+_d)[push(ref(ints), as<int>(_))];
```

You'll notice that doing it this way causes member function invocations to look like ordinary function invocations. You can choose to write your semantic action in a different way that makes it look a bit more like a member function call:

```
sregex rex = (+_d)[ref(ints)->*push(as<int>(_))];
```

Xpressive recognizes the use of the `->*` and treats this expression exactly the same as the one above.

When your function object must return a type that depends on its arguments, you can use a `result<>` member template instead of the `result_type` typedef. Here, for example, is a `first` function object that returns the `first` member of a `std::pair<>` or `sub_match<>`:

```
// Function object that returns the
// first element of a pair.
struct first_impl
{
    template<typename Sig> struct result {};

    template<typename This, typename Pair>
    struct result<This(Pair)>
    {
        typedef typename remove_reference<Pair>
            ::type::first_type type;
    };

    template<typename Pair>
    typename Pair::first_type
    operator()(Pair const &p) const
    {
        return p.first;
    }
};

// OK, use as first(s1) to get the begin iterator
// of the sub-match referred to by s1.
function<first_impl>::type const first = {{}};
```

Referring to Local Variables

As we've seen in the examples above, we can refer to local variables within an actions using `xpressive::ref()`. Any such variables are held by reference by the regular expression, and care should be taken to avoid letting those references dangle. For instance, in the following code, the reference to `i` is left to dangle when `bad_voodoo()` returns:

```
sregex bad_voodoo()
{
    int i = 0;
    sregex rex = +( _d [ ++ref(i) ] >> '!' );
    // ERROR! rex refers by reference to a local
    // variable, which will dangle after bad_voodoo()
    // returns.
    return rex;
}
```

When writing semantic actions, it is your responsibility to make sure that all the references do not dangle. One way to do that would be to make the variables shared pointers that are held by the regex by value.

```
sregex good_voodoo(boost::shared_ptr<int> pi)
{
    // Use val() to hold the shared_ptr by value:
    sregex rex = +( _d [ ++*val(pi) ] >> '!' );
    // OK, rex holds a reference count to the integer.
    return rex;
}
```

In the above code, we use `xpressive::val()` to hold the shared pointer by value. That's not normally necessary because local variables appearing in actions are held by value by default, but in this case, it is necessary. Had we written the action as `++*pi`, it would have executed immediately. That's because `++*pi` is not an expression template, but `++*val(pi)` is.

It can be tedious to wrap all your variables in `ref()` and `val()` in your semantic actions. Xpressive provides the `reference<>` and `value<>` templates to make things easier. The following table shows the equivalencies:

Table 12. reference<> and value<>

This ...	... is equivalent to this ...

`int i = 0;`
 	
`sregex rex = +( _d [ ++ref(i) ] >> '!' );`
	`int i = 0;`
	`reference<int> ri(i);`
	`sregex rex = +( _d [ ++ri ] >> '!' );`

`boost::shared_ptr<int> pi(new int(0));`
 	
`sregex rex = +( _d [ ++*val(pi) ] >> '!' );`
	`boost::shared_ptr<int> pi(new int(0));`
	`value<boost::shared_ptr<int> > vpi(pi);`
	`sregex rex = +( _d [ ++*vpi ] >> '!' );`

As you can see, when using `reference<>`, you need to first declare a local variable and then declare a `reference<>` to it. These two steps can be combined into one using `local<>`.

Table 13. local<> vs. reference<>

This ...	... is equivalent to this ...

`local<int> i(0);`
 	
`sregex rex = +( _d [ ++i ] >> '!' );`
	`int i = 0;`
	`reference<int> ri(i);`
	`sregex rex = +( _d [ ++ri ] >> '!' );`

We can use `local<>` to rewrite the above example as follows:

```
local<int> i(0);
std::string str("1!2!3?");
// count the exciting digits, but not the
// questionable ones.
sregex rex = +( _d [ ++i ] >> '!' );
regex_search(str, rex);
assert( i.get() == 2 );
```

Notice that we use `local<>::get()` to access the value of the local variable. Also, beware that `local<>` can be used to create a dangling reference, just as `reference<>` can.

Referring to Non-Local Variables

In the beginning of this section, we used a regex with a semantic action to parse a string of word/integer pairs and stuff them into a `std::map<>`. That required that the map and the regex be defined together and used before either could go out of scope. What if we wanted to define the regex once and use it to fill lots of different maps? We would rather pass the map into the `regex_match()` algorithm rather than embed a reference to it directly in the regex object. What we can do instead is define a placeholder and use that in the semantic action instead of the map itself. Later, when we call one of the regex algorithms, we can bind the reference to an actual map object. The following code shows how.

```
// Define a placeholder for a map object:
placeholder<std::map<std::string, int> > _map;

// Match a word and an integer, separated by =>,
// and then stuff the result into a std::map<>
sregex pair = ( (s1= +_w) >> "=>" >> (s2= +_d) )
    [ _map[s1] = as<int>(s2) ];

// Match one or more word/integer pairs, separated
// by whitespace.
sregex rx = pair >> *(+_s >> pair);

// The string to parse
std::string str("aaa=>1 bbb=>23 ccc=>456");

// Here is the actual map to fill in:
std::map<std::string, int> result;

// Bind the _map placeholder to the actual map
smatch what;
what.let( _map = result );

// Execute the match and fill in result map
if(regex_match(str, what, rx))
{
    std::cout << result["aaa"] << '\n';
    std::cout << result["bbb"] << '\n';
    std::cout << result["ccc"] << '\n';
}
```

This program displays:

```
1
23
456
```

We use `placeholder<>` here to define `_map`, which stands in for a `std::map<>` variable. We can use the placeholder in the semantic action as if it were a map. Then, we define a `match_results<>` struct and bind an actual map to the placeholder with "`what.let( _map = result );`". The `regex_match()` call behaves as if the placeholder in the semantic action had been replaced with a reference to `result`.

Note

Placeholders in semantic actions are not *actually* replaced at runtime with references to variables. The regex object is never mutated in any way during any of the regex algorithms, so they are safe to use in multiple threads.

The syntax for late-bound action arguments is a little different if you are using `regex_iterator<>` or `regex_token_iterator<>`. The regex iterators accept an extra constructor parameter for specifying the argument bindings. There is a `let()` function that you can use to bind variables to their placeholders. The following code demonstrates how.

```
// Define a placeholder for a map object:
placeholder<std::map<std::string, int> > _map;

// Match a word and an integer, separated by =>,
// and then stuff the result into a std::map<>
sregex pair = ( (s1= +_w) >> "=>" >> (s2= +_d) )
    [ _map[s1] = as<int>(s2) ];

// The string to parse
std::string str("aaa=>1 bbb=>23 ccc=>456");

// Here is the actual map to fill in:
std::map<std::string, int> result;

// Create a regex_iterator to find all the matches
sregex_iterator it(str.begin(), str.end(), pair, let(_map=result));
sregex_iterator end;

// step through all the matches, and fill in
// the result map
while(it != end)
    ++it;

std::cout << result["aaa"] << '\n';
std::cout << result["bbb"] << '\n';
std::cout << result["ccc"] << '\n';
```

This program displays:

```
1
23
456
```

User-Defined Assertions

You are probably already familiar with regular expression *assertions*. In Perl, some examples are the ^ and $ assertions, which you can use to match the beginning and end of a string, respectively. Xpressive lets you define your own assertions. A custom assertion is a contition which must be true at a point in the match in order for the match to succeed. You can check a custom assertion with xpressive's `check()` function.

There are a couple of ways to define a custom assertion. The simplest is to use a function object. Let's say that you want to ensure that a sub-expression matches a sub-string that is either 3 or 6 characters long. The following struct defines such a predicate:

```
// A predicate that is true IFF a sub-match is
// either 3 or 6 characters long.
struct three_or_six
{
    bool operator()(ssub_match const &sub) const
    {
        return sub.length() == 3 || sub.length() == 6;
    }
};
```

You can use this predicate within a regular expression as follows:

```
// match words of 3 characters or 6 characters.
sregex rx = (bow >> +_w >> eow)[ check(three_or_six()) ] ;
```

The above regular expression will find whole words that are either 3 or 6 characters long. The `three_or_six` predicate accepts a `sub_match<>` that refers back to the part of the string matched by the sub-expression to which the custom assertion is attached.

Note

The custom assertion participates in determining whether the match succeeds or fails. Unlike actions, which execute lazily, custom assertions execute immediately while the regex engine is searching for a match.

Custom assertions can also be defined inline using the same syntax as for semantic actions. Below is the same custom assertion written inline:

```
// match words of 3 characters or 6 characters.
sregex rx = (bow >> +_w >> eow)[ check(length(_)==3 || length(_)==6) ] ;
```

In the above, `length()` is a lazy function that calls the `length()` member function of its argument, and _ is a placeholder that receives the `sub_match`.

Once you get the hang of writing custom assertions inline, they can be very powerful. For example, you can write a regular expression that only matches valid dates (for some suitably liberal definition of the term “valid”).

```
int const days_per_month[] =
    {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 31, 31};

mark_tag month(1), day(2);
// find a valid date of the form month/day/year.
sregex date =
    (
        // Month must be between 1 and 12 inclusive
        (month= _d >> !_d)     [ check(as<int>(_) >= 1
                                    && as<int>(_) <= 12) ]
    >>  '/'
        // Day must be between 1 and 31 inclusive
    >>  (day=   _d >> !_d)     [ check(as<int>(_) >= 1
                                    && as<int>(_) <= 31) ]
    >>  '/'
        // Only consider years between 1970 and 2038
    >>  (_d >> _d >> _d >> _d) [ check(as<int>(_) >= 1970
                                    && as<int>(_) <= 2038) ]
    )
    // Ensure the month actually has that many days!
    [ check( ref(days_per_month)[as<int>(month)-1] >= as<int>(day) ) ]
;

smatch what;
std::string str("99/99/9999 2/30/2006 2/28/2006");

if(regex_search(str, what, date))
{
    std::cout << what[0] << std::endl;
}
```

The above program prints out the following:

```
2/28/2006
```

Notice how the inline custom assertions are used to range-check the values for the month, day and year. The regular expression doesn't match `"99/99/9999"` or `"2/30/2006"` because they are not valid dates. (There is no 99th month, and February doesn't have 30 days.)

Symbol Tables and Attributes

Overview

Symbol tables can be built into xpressive regular expressions with just a `std::map<>`. The map keys are the strings to be matched and the map values are the data to be returned to your semantic action. Xpressive attributes, named `a1`, `a2`, through `a9`, hold the value corresponding to a matching key so that it can be used in a semantic action. A default value can be specified for an attribute if a symbol is not found.

Symbol Tables

An xpressive symbol table is just a `std::map<>`, where the key is a string type and the value can be anything. For example, the following regular expression matches a key from map1 and assigns the corresponding value to the attribute `a1`. Then, in the semantic action, it assigns the value stored in attribute `a1` to an integer result.

```
int result;
std::map<std::string, int> map1;
// ... (fill the map)
sregex rx = ( a1 = map1 ) [ ref(result) = a1 ];
```

Consider the following example code, which translates number names into integers. It is described below.

```
#include <string>
#include <iostream>
#include <boost/xpressive/xpressive.hpp>
#include <boost/xpressive/regex_actions.hpp>
using namespace boost::xpressive;

int main()
{
    std::map<std::string, int> number_map;
    number_map["one"] = 1;
    number_map["two"] = 2;
    number_map["three"] = 3;
    // Match a string from number_map
    // and store the integer value in 'result'
    // if not found, store -1 in 'result'
    int result = 0;
    cregex rx = ((a1 = number_map ) | *_)
        [ ref(result) = (a1 | -1)];

    regex_match("three", rx);
    std::cout << result << '\n';
    regex_match("two", rx);
    std::cout << result << '\n';
    regex_match("stuff", rx);
    std::cout << result << '\n';
    return 0;
}
```

This program prints the following:

```
3
2
-1
```

First the program builds a number map, with number names as string keys and the corresponding integers as values. Then it constructs a static regular expression using an attribute `a1` to represent the result of the symbol table lookup. In the semantic action, the attribute

is assigned to an integer variable `result`. If the symbol was not found, a default value of `-1` is assigned to `result`. A wildcard, `*_`, makes sure the regex matches even if the symbol is not found.

A more complete version of this example can be found in `libs/xpressive/example/numbers.cpp`[2]. It translates number names up to "nine hundred ninety nine million nine hundred ninety nine thousand nine hundred ninety nine" along with some special number names like "dozen".

Symbol table matches are case sensitive by default, but they can be made case-insensitive by enclosing the expression in `icase()`.

Attributes

Up to nine attributes can be used in a regular expression. They are named `a1`, `a2`, ..., `a9` in the `boost::xpressive` namespace. The attribute type is the same as the second component of the map that is assigned to it. A default value for an attribute can be specified in a semantic action with the syntax `(a1 | default-value)`.

Attributes are properly scoped, so you can do crazy things like: `( (a1=sym1) >> (a1=sym2)[ref(x)=a1] )[ref(y)=a1]`. The inner semantic action sees the inner `a1`, and the outer semantic action sees the outer one. They can even have different types.

Note

Xpressive builds a hidden ternary search trie from the map so it can search quickly. If BOOST_DISABLE_THREADS is defined, the hidden ternary search trie "self adjusts", so after each search it restructures itself to improve the efficiency of future searches based on the frequency of previous searches.

Localization and Regex Traits

Overview

Matching a regular expression against a string often requires locale-dependent information. For example, how are case-insensitive comparisons performed? The locale-sensitive behavior is captured in a traits class. xpressive provides three traits class templates: `cpp_regex_traits<>`, `c_regex_traits<>` and `null_regex_traits<>`. The first wraps a `std::locale`, the second wraps the global C locale, and the third is a stub traits type for use when searching non-character data. All traits templates conform to the Regex Traits Concept.

Setting the Default Regex Trait

By default, xpressive uses `cpp_regex_traits<>` for all patterns. This causes all regex objects to use the global `std::locale`. If you compile with `BOOST_XPRESSIVE_USE_C_TRAITS` defined, then xpressive will use `c_regex_traits<>` by default.

Using Custom Traits with Dynamic Regexes

To create a dynamic regex that uses a custom traits object, you must use `regex_compiler<>`. The basic steps are shown in the following example:

```
// Declare a regex_compiler that uses the global C locale
regex_compiler<char const *, c_regex_traits<char> > crxcomp;
cregex crx = crxcomp.compile( "\\w+" );

// Declare a regex_compiler that uses a custom std::locale
std::locale loc = /* ... create a locale here ... */;
regex_compiler<char const *, cpp_regex_traits<char> > cpprxcomp(loc);
cregex cpprx = cpprxcomp.compile( "\\w+" );
```

[2] Many thanks to David Jenkins, who contributed this example.

The `regex_compiler` objects act as regex factories. Once they have been imbued with a locale, every regex object they create will use that locale.

Using Custom Traits with Static Regexes

If you want a particular static regex to use a different set of traits, you can use the special `imbue()` pattern modifier. For instance:

```
// Define a regex that uses the global C locale
c_regex_traits<char> ctraits;
sregex crx = imbue(ctraits)( +_w );

// Define a regex that uses a customized std::locale
std::locale loc = /* ... create a locale here ... */;
cpp_regex_traits<char> cpptraits(loc);
sregex cpprx1 = imbue(cpptraits)( +_w );

// A shorthand for above
sregex cpprx2 = imbue(loc)( +_w );
```

The `imbue()` pattern modifier must wrap the entire pattern. It is an error to `imbue` only part of a static regex. For example:

```
// ERROR! Cannot imbue() only part of a regex
sregex error = _w >> imbue(loc)( _w );
```

Searching Non-Character Data With `null_regex_traits`

With xpressive static regexes, you are not limitted to searching for patterns in character sequences. You can search for patterns in raw bytes, integers, or anything that conforms to the Char Concept. The `null_regex_traits<>` makes it simple. It is a stub implementation of the Regex Traits Concept. It recognizes no character classes and does no case-sensitive mappings.

For example, with `null_regex_traits<>`, you can write a static regex to find a pattern in a sequence of integers as follows:

```
// some integral data to search
int const data[] = {0, 1, 2, 3, 4, 5, 6};

// create a null_regex_traits<> object for searching integers ...
null_regex_traits<int> nul;

// imbue a regex object with the null_regex_traits ...
basic_regex<int const *> rex = imbue(nul)(1 >> +((set= 2,3) | 4) >> 5);
match_results<int const *> what;

// search for the pattern in the array of integers ...
regex_search(data, data + 7, what, rex);

assert(what[0].matched);
assert(*what[0].first == 1);
assert(*what[0].second == 6);
```

Tips 'N Tricks

Squeeze the most performance out of xpressive with these tips and tricks.

Compile Patterns Once And Reuse Them

Compiling a regex (dynamic or static) is *far* more expensive than executing a match or search. If you have the option, prefer to compile a pattern into a `basic_regex<>` object once and reuse it rather than recreating it over and over.

Since `basic_regex<>` objects are not mutated by any of the regex algorithms, they are completely thread-safe once their initialization (and that of any grammars of which they are members) completes. The easiest way to reuse your patterns is to simply make your `basic_regex<>` objects "static const".

Reuse `match_results<>` Objects

The `match_results<>` object caches dynamically allocated memory. For this reason, it is far better to reuse the same `match_results<>` object if you have to do many regex searches.

Caveat: `match_results<>` objects are not thread-safe, so don't go wild reusing them across threads.

Prefer Algorithms That Take A `match_results<>` Object

This is a corollary to the previous tip. If you are doing multiple searches, you should prefer the regex algorithms that accept a `match_results<>` object over the ones that don't, and you should reuse the same `match_results<>` object each time. If you don't provide a `match_results<>` object, a temporary one will be created for you and discarded when the algorithm returns. Any memory cached in the object will be deallocated and will have to be reallocated the next time.

Prefer Algorithms That Accept Iterator Ranges Over Null-Terminated Strings

xpressive provides overloads of the `regex_match()` and `regex_search()` algorithms that operate on C-style null-terminated strings. You should prefer the overloads that take iterator ranges. When you pass a null-terminated string to a regex algorithm, the end iterator is calculated immediately by calling `strlen`. If you already know the length of the string, you can avoid this overhead by calling the regex algorithms with a `[begin, end)` pair.

Use Static Regexes

On average, static regexes execute about 10 to 15% faster than their dynamic counterparts. It's worth familiarizing yourself with the static regex dialect.

Understand `syntax_option_type::optimize`

The `optimize` flag tells the regex compiler to spend some extra time analyzing the pattern. It can cause some patterns to execute faster, but it increases the time to compile the pattern, and often increases the amount of memory consumed by the pattern. If you plan to reuse your pattern, `optimize` is usually a win. If you will only use the pattern once, don't use `optimize`.

Common Pitfalls

Keep the following tips in mind to avoid stepping in potholes with xpressive.

Create Grammars On A Single Thread

With static regexes, you can create grammars by nesting regexes inside one another. When compiling the outer regex, both the outer and inner regex objects, and all the regex objects to which they refer either directly or indirectly, are modified. For this reason, it's dangerous for global regex objects to participate in grammars. It's best to build regex grammars from a single thread. Once built, the resulting regex grammar can be executed from multiple threads without problems.

Beware Nested Quantifiers

This is a pitfall common to many regular expression engines. Some patterns can cause exponentially bad performance. Often these patterns involve one quantified term nested withing another quantifier, such as `"(a*)*"`, although in many cases, the problem is harder to spot. Beware of patterns that have nested quantifiers.

Concepts

CharT requirements

If type `BidiIterT` is used as a template argument to `basic_regex<>`, then `CharT` is `iterator_traits<BidiIterT>::value_type`. Type `CharT` must have a trivial default constructor, copy constructor, assignment operator, and destructor. In addition the following requirements must be met for objects; `c` of type `CharT`, `c1` and `c2` of type `CharT const`, and `i` of type `int`:

Table 14. CharT Requirements

Expression	Return type	Assertion / Note / Pre- / Post-condition
`CharT c`	`CharT`	Default constructor (must be trivial).
`CharT c(c1)`	`CharT`	Copy constructor (must be trivial).
`c1 = c2`	`CharT`	Assignment operator (must be trivial).
`c1 == c2`	`bool`	`true` if `c1` has the same value as `c2`.
`c1 != c2`	`bool`	`true` if `c1` and `c2` are not equal.
`c1 < c2`	`bool`	`true` if the value of `c1` is less than `c2`.
`c1 > c2`	`bool`	`true` if the value of `c1` is greater than `c2`.
`c1 <= c2`	`bool`	`true` if `c1` is less than or equal to `c2`.
`c1 >= c2`	`bool`	`true` if `c1` is greater than or equal to `c2`.
`intmax_t i = c1`	`int`	`CharT` must be convertible to an integral type.
`CharT c(i);`	`CharT`	`CharT` must be constructable from an integral type.

Traits Requirements

In the following table `X` denotes a traits class defining types and functions for the character container type `CharT`; `u` is an object of type `X`; `v` is an object of type `const X`; `p` is a value of type `const CharT*`; `I1` and `I2` are `Input Iterators`; `c` is a value of type `const CharT`; `s` is an object of type `X::string_type`; `cs` is an object of type `const X::string_type`; `b` is a value of type `bool`; `i` is a value of type `int`; `F1` and `F2` are values of type `const CharT*`; `loc` is an object of type `X::locale_type`; and `ch` is an object of `const char`.

Table 15. Traits Requirements

Expression	Return type	Assertion / Note Pre / Post condition
`X::char_type`	`CharT`	The character container type used in the implementation of class template `basic_regex<>`.
`X::string_type`	`std::basic_string<CharT>` or `std::vector<CharT>`	
`X::locale_type`	*Implementation defined*	A copy constructible type that represents the locale used by the traits class.
`X::char_class_type`	*Implementation defined*	A bitmask type representing a particular character classification. Multiple values of this type can be bitwise-or'ed together to obtain a new valid value.
`X::hash(c)`	`unsigned char`	Yields a value between `0` and `UCHAR_MAX` inclusive.
`v.widen(ch)`	`CharT`	Widens the specified `char` and returns the resulting `CharT`.
`v.in_range(r1, r2, c)`	`bool`	For any characters `r1` and `r2`, returns `true` if `r1 <= c && c <= r2`. Requires that `r1 <= r2`.
`v.in_range_nocase(r1, r2, c)`	`bool`	For characters `r1` and `r2`, returns `true` if there is some character `d` for which `v.translate_nocase(d) == v.translate_nocase(c)` and `r1 <= d && d <= r2`. Requires that `r1 <= r2`.
`v.translate(c)`	`X::char_type`	Returns a character such that for any character `d` that is to be considered equivalent to `c` then `v.translate(c) == v.translate(d)`.
`v.translate_nocase(c)`	`X::char_type`	For all characters `C` that are to be considered equivalent to `c` when comparisons are to be performed without regard to case, then `v.translate_nocase(c) == v.translate_nocase(C)`.
`v.transform(F1, F2)`	`X::string_type`	Returns a sort key for the character sequence designated by the iterator range `[F1, F2)` such that if the character sequence `[G1, G2)` sorts before the character sequence `[H1, H2)` then `v.transform(G1, G2) < v.transform(H1, H2)`.

Expression	Return type	Assertion / Note Pre / Post condition
`v.transform_primary(F1, F2)`	`X::string_type`	Returns a sort key for the character sequence designated by the iterator range `[F1, F2)` such that if the character sequence `[G1, G2)` sorts before the character sequence `[H1, H2)` when character case is not considered then `v.transform_primary(G1, G2) < v.transform_primary(H1, H2)`.
`v.lookup_classname(F1, F2)`	`X::char_class_type`	Converts the character sequence designated by the iterator range `[F1,F2)` into a bitmask type that can subsequently be passed to `isctype`. Values returned from `lookup_classname` can be safely bitwise or'ed together. Returns `0` if the character sequence is not the name of a character class recognized by `X`. The value returned shall be independent of the case of the characters in the sequence.
`v.lookup_collatename(F1, F2)`	`X::string_type`	Returns a sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range `[F1, F2)`. Returns an empty string if the character sequence is not a valid collating element.
`v.isctype(c, v.lookup_classname(F1, F2))`	`bool`	Returns `true` if character `c` is a member of the character class designated by the iterator range `[F1, F2)`, `false` otherwise.
`v.value(c, i)`	`int`	Returns the value represented by the digit `c` in base `i` if the character `c` is a valid digit in base `i`; otherwise returns `-1`. [Note: the value of `i` will only be `8`, `10`, or `16`. -end note]
`u.imbue(loc)`	`X::locale_type`	Imbues `u` with the locale `loc`, returns the previous locale used by `u`.
`v.getloc()`	`X::locale_type`	Returns the current locale used by `v`.

Acknowledgements

This section is adapted from the equivalent page in the Boost.Regex documentation and from the proposal to add regular expressions to the Standard Library.

Examples

Below you can find six complete sample programs.

See if a whole string matches a regex

This is the example from the Introduction. It is reproduced here for your convenience.

```
#include <iostream>
#include <boost/xpressive/xpressive.hpp>

using namespace boost::xpressive;

int main()
{
    std::string hello( "hello world!" );

    sregex rex = sregex::compile( "(\\w+) (\\w+)!" );
    smatch what;

    if( regex_match( hello, what, rex ) )
    {
        std::cout << what[0] << '\n'; // whole match
        std::cout << what[1] << '\n'; // first capture
        std::cout << what[2] << '\n'; // second capture
    }

    return 0;
}
```

This program outputs the following:

```
hello world!
hello
world
```

top

See if a string contains a sub-string that matches a regex

Notice in this example how we use custom `mark_tag`s to make the pattern more readable. We can use the `mark_tag`s later to index into the `match_results<>`.

```
#include <iostream>
#include <boost/xpressive/xpressive.hpp>

using namespace boost::xpressive;

int main()
{
    char const *str = "I was born on 5/30/1973 at 7am.";

    // define some custom mark_tags with names more meaningful than s1, s2, etc.
    mark_tag day(1), month(2), year(3), delim(4);

    // this regex finds a date
    cregex date = (month= repeat<1,2>(_d))           // find the month ...
                >> (delim= (set= '/','-'))           // followed by a delimiter ...
              >> (day=   repeat<1,2>(_d)) >> delim  // and a day followed by the same delimiter ↵
...
                >> (year=  repeat<1,2>(_d >> _d));    // and the year.

    cmatch what;

    if( regex_search( str, what, date ) )
    {
        std::cout << what[0]     << '\n'; // whole match
        std::cout << what[day]   << '\n'; // the day
        std::cout << what[month] << '\n'; // the month
        std::cout << what[year]  << '\n'; // the year
        std::cout << what[delim] << '\n'; // the delimiter
    }

    return 0;
}
```

This program outputs the following:

```
5/30/1973
30
5
1973
/
```

top

Replace all sub-strings that match a regex

The following program finds dates in a string and marks them up with pseudo-HTML.

```
#include <iostream>
#include <boost/xpressive/xpressive.hpp>

using namespace boost::xpressive;

int main()
{
    std::string str( "I was born on 5/30/1973 at 7am." );

    // essentially the same regex as in the previous example, but using a dynamic regex
    sregex date = sregex::compile( "(\\d{1,2})([/-])(\\d{1,2})\\2((?:\\d{2}){1,2})" );

    // As in Perl, $& is a reference to the sub-string that matched the regex
    std::string format( "<date>$&</date>" );

    str = regex_replace( str, date, format );
    std::cout << str << '\n';

    return 0;
}
```

This program outputs the following:

```
I was born on <date>5/30/1973</date> at 7am.
```

top

Find all the sub-strings that match a regex and step through them one at a time

The following program finds the words in a wide-character string. It uses `wsregex_iterator`. Notice that dereferencing a `wsregex_iterator` yields a `wsmatch` object.

```
#include <iostream>
#include <boost/xpressive/xpressive.hpp>

using namespace boost::xpressive;

int main()
{
    std::wstring str( L"This is his face." );

    // find a whole word
    wsregex token = +alnum;

    wsregex_iterator cur( str.begin(), str.end(), token );
    wsregex_iterator end;

    for( ; cur != end; ++cur )
    {
        wsmatch const &what = *cur;
        std::wcout << what[0] << L'\n';
    }

    return 0;
}
```

This program outputs the following:

```
This
is
his
face
```

top

Split a string into tokens that each match a regex

The following program finds race times in a string and displays first the minutes and then the seconds. It uses `regex_token_iterator<>`.

```
#include <iostream>
#include <boost/xpressive/xpressive.hpp>

using namespace boost::xpressive;

int main()
{
    std::string str( "Eric: 4:40, Karl: 3:35, Francesca: 2:32" );

    // find a race time
    sregex time = sregex::compile( "(\\d):(\\d\\d)" );

    // for each match, the token iterator should first take the value of
    // the first marked sub-expression followed by the value of the second
    // marked sub-expression
    int const subs[] = { 1, 2 };

    sregex_token_iterator cur( str.begin(), str.end(), time, subs );
    sregex_token_iterator end;

    for( ; cur != end; ++cur )
    {
        std::cout << *cur << '\n';
    }

    return 0;
}
```

This program outputs the following:

```
4
40
3
35
2
32
```

top

Split a string using a regex as a delimiter

The following program takes some text that has been marked up with html and strips out the mark-up. It uses a regex that matches an HTML tag and a `regex_token_iterator<>` that returns the parts of the string that do *not* match the regex.

```
#include <iostream>
#include <boost/xpressive/xpressive.hpp>

using namespace boost::xpressive;

int main()
{
    std::string str( "Now <bold>is the time <i>for all good men</i> to come to the aid of ↵
their</bold> country." );

    // find a HTML tag
    sregex html = '<' >> optional('/') >> +_w >> '>';

    // the -1 below directs the token iterator to display the parts of
    // the string that did NOT match the regular expression.
    sregex_token_iterator cur( str.begin(), str.end(), html, -1 );
    sregex_token_iterator end;

    for( ; cur != end; ++cur )
    {
        std::cout << '{' << *cur << '}';
    }
    std::cout << '\n';

    return 0;
}
```

This program outputs the following:

```
{Now }{is the time }{for all good men}{ to come to the aid of their}{ country.}
```

top

Display a tree of nested results

Here is a helper class to demonstrate how you might display a tree of nested results:

```
// Displays nested results to std::cout with indenting
struct output_nested_results
{
    int tabs_;

    output_nested_results( int tabs = 0 )
        : tabs_( tabs )
    {
    }

    template< typename BidiIterT >
    void operator ()( match_results< BidiIterT > const &what ) const
    {
        // first, do some indenting
        typedef typename std::iterator_traits< BidiIterT >::value_type char_type;
        char_type space_ch = char_type(' ');
        std::fill_n( std::ostream_iterator<char_type>( std::cout ), tabs_ * 4, space_ch );

        // output the match
        std::cout << what[0] << '\n';

        // output any nested matches
        std::for_each(
            what.nested_results().begin(),
            what.nested_results().end(),
            output_nested_results( tabs_ + 1 ) );
    }
};
```

top

Reference

Header <boost/xpressive/basic_regex.hpp>

Contains the definition of the basic_regex<> class template and its associated helper functions.

```
namespace boost {
  namespace xpressive {
    template<typename BidiIter> struct basic_regex;
    template<typename BidiIter>
      void swap(basic_regex< BidiIter > &, basic_regex< BidiIter > &);
  }
}
```

Struct template basic_regex

boost::xpressive::basic_regex — Class template basic_regex<> is a class for holding a compiled regular expression.

Synopsis

```
// In header: <boost/xpressive/basic_regex.hpp>

template<typename BidiIter>
struct basic_regex {
  // types
  typedef BidiIter                              iterator_type;
  typedef iterator_value< BidiIter >::type      char_type;
  typedef iterator_value< BidiIter >::type      value_type;
  typedef unspecified                           string_type;
  typedef regex_constants::syntax_option_type flag_type;

  // construct/copy/destruct
  basic_regex();
  basic_regex(basic_regex< BidiIter > const &);
  template<typename Expr> basic_regex(Expr const &);
  basic_regex< BidiIter > & operator=(basic_regex< BidiIter > const &);
  template<typename Expr> basic_regex< BidiIter > & operator=(Expr const &);

  // public member functions
  std::size_t mark_count() const;
  regex_id_type regex_id() const;
  void swap(basic_regex< BidiIter > &);

  // public static functions
  template<typename InputIter>
    static basic_regex< BidiIter >
    compile(InputIter, InputIter, flag_type = regex_constants::ECMAScript);
  template<typename InputRange>
    static basic_regex< BidiIter >
    compile(InputRange const &, flag_type = regex_constants::ECMAScript);
  static basic_regex< BidiIter >
  compile(char_type const *, flag_type = regex_constants::ECMAScript);
  static basic_regex< BidiIter >
  compile(char_type const *, std::size_t, flag_type);

  // public data members
  static regex_constants::syntax_option_type const ECMAScript;
  static regex_constants::syntax_option_type const icase;
  static regex_constants::syntax_option_type const nosubs;
  static regex_constants::syntax_option_type const optimize;
  static regex_constants::syntax_option_type const collate;
  static regex_constants::syntax_option_type const single_line;
  static regex_constants::syntax_option_type const not_dot_null;
  static regex_constants::syntax_option_type const not_dot_newline;
  static regex_constants::syntax_option_type const ignore_white_space;
};
```

Description

`basic_regex` public construct/copy/destruct

1. ```
   basic_regex();
   ```

   Postconditions: regex_id() == 0
   Postconditions: mark_count() == 0

2. ```
   basic_regex(basic_regex< BidiIter > const & that);
   ```

Parameters: `that` The `basic_regex` object to copy.
Postconditions: regex_id() == that.regex_id()
Postconditions: mark_count() == that.mark_count()

3.
```
template<typename Expr> basic_regex(Expr const & expr);
```

Construct from a static regular expression.

Parameters: `expr` The static regular expression
Requires: Expr is the type of a static regular expression.
Postconditions: regex_id() != 0
Postconditions: mark_count() >= 0

4.
```
basic_regex< BidiIter > & operator=(basic_regex< BidiIter > const & that);
```

Parameters: `that` The `basic_regex` object to copy.
Postconditions: regex_id() == that.regex_id()
Postconditions: mark_count() == that.mark_count()
Returns: *this

5.
```
template<typename Expr> basic_regex< BidiIter > & operator=(Expr const & expr);
```

Construct from a static regular expression.

Parameters: `expr` The static regular expression.
Requires: Expr is the type of a static regular expression.
Postconditions: regex_id() != 0
Postconditions: mark_count() >= 0
Returns: *this
Throws: std::bad_alloc on out of memory

`basic_regex` public member functions

1.
```
std::size_t mark_count() const;
```

Returns the count of capturing sub-expressions in this regular expression

2.
```
regex_id_type regex_id() const;
```

Returns a token which uniquely identifies this regular expression.

3.
```
void swap(basic_regex< BidiIter > & that);
```

Swaps the contents of this `basic_regex` object with another.

Note

This is a shallow swap that does not do reference tracking. If you embed a `basic_regex` object by reference in another regular expression and then swap its contents with another `basic_regex` object, the change will not be visible to the enclosing regular expression. It is done this way to ensure that swap() cannot throw.

Parameters: `that` The other `basic_regex` object.
Throws: Will not throw.

`basic_regex` public static functions

1.
```
template<typename InputIter>
  static basic_regex< BidiIter >
  compile(InputIter begin, InputIter end,
          flag_type flags = regex_constants::ECMAScript);
```

Factory method for building a regex object from a range of characters. Equivalent to `regex_compiler`< BidiIter >().compile(begin, end, flags);

Parameters:	`begin`	The beginning of a range of characters representing the regular expression to compile.
	`end`	The end of a range of characters representing the regular expression to compile.
	`flags`	Optional bitmask that determines how the pat string is interpreted. (See syntax_option_type.)
Requires:	[begin,end) is a valid range.	
Requires:	The range of characters specified by [begin,end) contains a valid string-based representation of a regular expression.	
Returns:	A basic_regex object corresponding to the regular expression represented by the character range.	
Throws:	regex_error when the range of characters has invalid regular expression syntax.	

2.
```
template<typename InputRange>
  static basic_regex< BidiIter >
  compile(InputRange const & pat,
          flag_type flags = regex_constants::ECMAScript);
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

3.
```
static basic_regex< BidiIter >
compile(char_type const * begin,
        flag_type flags = regex_constants::ECMAScript);
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

4.
```
static basic_regex< BidiIter >
compile(char_type const * begin, std::size_t len, flag_type flags);
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Function template swap

boost::xpressive::swap — Swaps the contents of two basic_regex objects.

Synopsis

```
// In header: <boost/xpressive/basic_regex.hpp>


template<typename BidiIter>
  void swap(basic_regex< BidiIter > & left, basic_regex< BidiIter > & right);
```

Description

Note

This is a shallow swap that does not do reference tracking. If you embed a `basic_regex` object by reference in another regular expression and then swap its contents with another `basic_regex` object, the change will not be visible to the enclosing regular expression. It is done this way to ensure that swap() cannot throw.

Parameters: `left` The first `basic_regex` object.
`right` The second `basic_regex` object.
Throws: Will not throw.

Header <boost/xpressive/match_results.hpp>

Contains the definition of the match_results type and associated helpers. The match_results type holds the results of a regex_match() or regex_search() operation.

```
namespace boost {
  namespace xpressive {
    template<typename BidiIter> struct match_results;
    template<typename BidiIter> struct regex_id_filter_predicate;
  }
}
```

Struct template match_results

boost::xpressive::match_results — Class template match_results<> holds the results of a regex_match() or a regex_search() as a collection of sub_match objects.

Synopsis

```
// In header: <boost/xpressive/match_results.hpp>

template<typename BidiIter>
struct match_results {
  // types
  typedef iterator_value< BidiIter >::type      char_type;
  typedef unspecified                           string_type;
  typedef std::size_t                           size_type;
  typedef sub_match< BidiIter >                 value_type;
  typedef iterator_difference< BidiIter >::type difference_type;
  typedef value_type const &                    reference;
  typedef value_type const &                    const_reference;
  typedef unspecified                           iterator;
  typedef unspecified                           const_iterator;
  typedef unspecified                           nested_results_type;

  // construct/copy/destruct
  match_results();
  match_results(match_results< BidiIter > const &);
  match_results< BidiIter > & operator=(match_results< BidiIter > const &);
  ~match_results();

  // public member functions
  size_type size() const;
  bool empty() const;
  difference_type length(size_type = 0) const;
  difference_type position(size_type = 0) const;
  string_type str(size_type = 0) const;
  template<typename Sub> const_reference operator[](Sub const &) const;
  const_reference prefix() const;
  const_reference suffix() const;
  const_iterator begin() const;
  const_iterator end() const;
  operator bool_type() const;
  bool operator!() const;
  regex_id_type regex_id() const;
  nested_results_type const & nested_results() const;
  template<typename Format, typename OutputIterator>
    OutputIterator
    format(OutputIterator, Format const &,
           regex_constants::match_flag_type = regex_constants::format_default,
           unspecified = 0) const;
  template<typename OutputIterator>
    OutputIterator
    format(OutputIterator, char_type const *,
           regex_constants::match_flag_type = regex_constants::format_default) const;
  template<typename Format, typename OutputIterator>
    string_type format(Format const &,
                       regex_constants::match_flag_type = regex_constants::format_default,
                       unspecified = 0) const;
  string_type format(char_type const *,
                     regex_constants::match_flag_type = regex_constants::format_default) const;
  void swap(match_results< BidiIter > &);
  template<typename Arg> match_results< BidiIter > & let(Arg const &);
};
```

Description

Class template match_results<> denotes a collection of sequences representing the result of a regular expression match. Storage for the collection is allocated and freed as necessary by the member functions of class match_results<>.

The class template match_results<> conforms to the requirements of a Sequence, as specified in (lib.sequence.reqmts), except that only operations defined for const-qualified Sequences are supported.

`match_results` public construct/copy/destruct

1.
```
match_results();
```

Postconditions: regex_id() == 0
Postconditions: size() == 0
Postconditions: empty() == true
Postconditions: str() == string_type()

2.
```
match_results(match_results< BidiIter > const & that);
```

Parameters: `that` The `match_results` object to copy
Postconditions: regex_id() == that.regex_id().
Postconditions: size() == that.size().
Postconditions: empty() == that.empty().
Postconditions: str(n) == that.str(n) for all positive integers n < that.size().
Postconditions: prefix() == that.prefix().
Postconditions: suffix() == that.suffix().
Postconditions: (*this)[n] == that[n] for all positive integers n < that.size().
Postconditions: length(n) == that.length(n) for all positive integers n < that.size().
Postconditions: position(n) == that.position(n) for all positive integers n < that.size().

3.
```
match_results< BidiIter > & operator=(match_results< BidiIter > const & that);
```

Parameters: `that` The `match_results` object to copy.
Postconditions: regex_id() == that.regex_id().
Postconditions: size() == that.size().
Postconditions: empty() == that.empty().
Postconditions: str(n) == that.str(n) for all positive integers n < that.size().
Postconditions: prefix() == that.prefix().
Postconditions: suffix() == that.suffix().
Postconditions: (*this)[n] == that[n] for all positive integers n < that.size().
Postconditions: length(n) == that.length(n) for all positive integers n < that.size().
Postconditions: position(n) == that.position(n) for all positive integers n < that.size().

4.
```
~match_results();
```

`match_results` public member functions

1.
```
size_type size() const;
```

Returns one plus the number of marked sub-expressions in the regular expression that was matched if *this represents the result of a successful match. Otherwise returns 0.

2.
```
bool empty() const;
```

Returns size() == 0.

3.
```
difference_type length(size_type sub = 0) const;
```

Returns (*this)[sub].length().

4.
```
difference_type position(size_type sub = 0) const;
```

If !(*this)[sub].matched then returns -1. Otherwise returns std::distance(base, (*this)[sub].first), where base is the start iterator of the sequence that was searched. [Note - unless this is part of a repeated search with a `regex_iterator` then base is the same as prefix().first - end note]

5.
```
string_type str(size_type sub = 0) const;
```

Returns (*this)[sub].str().

6.
```
template<typename Sub> const_reference operator[](Sub const & sub) const;
```

Returns a reference to the `sub_match` object representing the sequence that matched marked sub-expression sub. If sub == 0 then returns a reference to a `sub_match` object representing the sequence that matched the whole regular expression. If sub >= size() then returns a `sub_match` object representing an unmatched sub-expression.

7.
```
const_reference prefix() const;
```

Returns a reference to the `sub_match` object representing the character sequence from the start of the string being matched/searched, to the start of the match found.

Requires: (*this)[0].matched is true

8.
```
const_reference suffix() const;
```

Returns a reference to the `sub_match` object representing the character sequence from the end of the match found to the end of the string being matched/searched.

Requires: (*this)[0].matched is true

9.
```
const_iterator begin() const;
```

Returns a starting iterator that enumerates over all the marked sub-expression matches stored in *this.

10.
```
const_iterator end() const;
```

Returns a terminating iterator that enumerates over all the marked sub-expression matches stored in *this.

11.
```
operator bool_type() const;
```

Returns a true value if (*this)[0].matched, else returns a false value.

12.
```
bool operator!() const;
```

Returns true if empty() || !(*this)[0].matched, else returns false.

13.
```
regex_id_type regex_id() const;
```

Returns the id of the `basic_regex` object most recently used with this `match_results` object.

14.
```
nested_results_type const & nested_results() const;
```

Returns a Sequence of nested `match_results` elements.

15.
```
template<typename Format, typename OutputIterator>
  OutputIterator
  format(OutputIterator out, Format const & fmt,
         regex_constants::match_flag_type flags = regex_constants::format_default,
         unspecified = 0) const;
```

If `Format` models `ForwardRange` or is a null-terminated string, this function copies the character sequence in `fmt` to `OutputIterator out`. For each format specifier or escape sequence in `fmt`, replace that sequence with either the character(s) it represents, or the sequence within `*this` to which it refers. The bitmasks specified in flags determines what format specifiers or escape sequences are recognized. By default, this is the format used by ECMA-262, ECMAScript Language Specification, Chapter 15 part 5.4.11 String.prototype.replace.

Otherwise, if `Format` models `Callable<match_results<BidiIter>, OutputIterator, regex_constants::match_flag_type>`, this function returns `fmt(*this, out, flags)`.

Otherwise, if `Format` models `Callable<match_results<BidiIter>, OutputIterator>`, this function returns `fmt(*this, out)`.

Otherwise, if `Format` models `Callable<match_results<BidiIter> >`, this function returns `std::copy(x.begin(), x.end(), out)`, where `x` is the result of calling `fmt(*this)`.

16.
```
template<typename OutputIterator>
  OutputIterator
  format(OutputIterator out, char_type const * fmt,
         regex_constants::match_flag_type flags = regex_constants::format_default) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

17.
```
template<typename Format, typename OutputIterator>
  string_type format(Format const & fmt,
                     regex_constants::match_flag_type flags = regex_constants::format_default,

                      unspecified = 0) const;
```

If `Format` models `ForwardRange` or is a null-terminated string, this function returns a copy of the character sequence `fmt`. For each format specifier or escape sequence in `fmt`, replace that sequence with either the character(s) it represents, or the sequence within `*this` to which it refers. The bitmasks specified in `flags` determines what format specifiers or escape sequences are recognized. By default this is the format used by ECMA-262, ECMAScript Language Specification, Chapter 15 part 5.4.11 String.prototype.replace.

Otherwise, if `Format` models `Callable<match_results<BidiIter>, OutputIterator, regex_constants::match_flag_type>`, this function returns a `string_type` object `x` populated by calling `fmt(*this, out, flags)`, where `out` is a `back_insert_iterator` into `x`.

Otherwise, if `Format` models `Callable<match_results<BidiIter>, OutputIterator>`, this function returns a `string_type` object `x` populated by calling `fmt(*this, out)`, where `out` is a `back_insert_iterator` into `x`.

Otherwise, if `Format` models `Callable<match_results<BidiIter> >`, this function returns `fmt(*this)`.

18.
```
string_type format(char_type const * fmt,
                   regex_constants::match_flag_type flags = regex_constants::format_default)
 const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

19.
```
void swap(match_results< BidiIter > & that);
```

Swaps the contents of two `match_results` objects. Guaranteed not to throw.

Parameters: `that` The `match_results` object to swap with.

Postconditions: *this contains the sequence of matched sub-expressions that were in that, that contains the sequence of matched sub-expressions that were in *this.

Throws: Will not throw.

20.
```
template<typename Arg> match_results< BidiIter > & let(Arg const & arg);
```

TODO document me

Struct template regex_id_filter_predicate

boost::xpressive::regex_id_filter_predicate

Synopsis

```
// In header: <boost/xpressive/match_results.hpp>

template<typename BidiIter>
struct regex_id_filter_predicate :
  public std::unary_function< match_results< BidiIter >, bool >
{
  // construct/copy/destruct
  regex_id_filter_predicate(regex_id_type);

  // public member functions
  bool operator()(match_results< BidiIter > const &) const;
};
```

Description

`regex_id_filter_predicate` public construct/copy/destruct

1.
```
regex_id_filter_predicate(regex_id_type regex_id);
```

`regex_id_filter_predicate` public member functions

1.
```
bool operator()(match_results< BidiIter > const & res) const;
```

Header <boost/xpressive/regex_actions.hpp>

Defines the syntax elements of xpressive's action expressions.

```
namespace boost {
  namespace xpressive {
    template<typename PolymorphicFunctionObject> struct function;
    template<typename T> struct local;
    template<typename T, int I = 0> struct placeholder;
    template<typename T> struct reference;
    template<typename T> struct value;

    function< op::at >::type const at;    // at is a lazy PolymorphicFunctionObject for index↵
ing into a sequence in an xpressive semantic action.
    function< op::push >::type const push;    // push is a lazy PolymorphicFunctionObject for ↵
pushing a value into a container in an xpressive semantic action.
    function< op::push_back >::type const push_back;    // push_back is a lazy PolymorphicFunc↵
tionObject for pushing a value into a container in an xpressive semantic action.
    function< op::push_front >::type const push_front;    // push_front is a lazy Polymorphic↵
FunctionObject for pushing a value into a container in an xpressive semantic action.
    function< op::pop >::type const pop;    // pop is a lazy PolymorphicFunctionObject for pop↵
ping the top element from a sequence in an xpressive semantic action.
    function< op::pop_back >::type const pop_back;    // pop_back is a lazy PolymorphicFunctionOb↵
ject for popping the back element from a sequence in an xpressive semantic action.
    function< op::pop_front >::type const pop_front;    // pop_front is a lazy PolymorphicFunc↵
tionObject for popping the front element from a sequence in an xpressive semantic action.
    function< op::top >::type const top;    // top is a lazy PolymorphicFunctionObject for ac↵
cessing the top element from a stack in an xpressive semantic action.
    function< op::back >::type const back;    // back is a lazy PolymorphicFunctionObject for ↵
fetching the back element of a sequence in an xpressive semantic action.
    function< op::front >::type const front;    // front is a lazy PolymorphicFunctionObject ↵
for fetching the front element of a sequence in an xpressive semantic action.
    function< op::first >::type const first;    // first is a lazy PolymorphicFunctionObject ↵
for accessing the first element of a std::pair<> in an xpressive semantic action.
    function< op::second >::type const second;    // second is a lazy PolymorphicFunctionObject ↵
for accessing the second element of a std::pair<> in an xpressive semantic action.
    function< op::matched >::type const matched;    // matched is a lazy PolymorphicFunctionOb↵
ject for accessing the matched member of a xpressive::sub_match<> in an xpressive semantic action.
    function< op::length >::type const length;    // length is a lazy PolymorphicFunctionObject ↵
for computing the length of a xpressive::sub_match<> in an xpressive semantic action.
    function< op::str >::type const str;    // str is a lazy PolymorphicFunctionObject for con↵
verting a xpressive::sub_match<> to a std::basic_string<> in an xpressive semantic action.
    function< op::insert >::type const insert;    // insert is a lazy PolymorphicFunctionObject ↵
for inserting a value or a range of values into a sequence in an xpressive semantic action.
    function< op::make_pair >::type const make_pair;    // make_pair is a lazy PolymorphicFunc↵
tionObject for making a std::pair<> in an xpressive semantic action.
    function< op::unwrap_reference >::type const unwrap_reference;    // unwrap_reference is a ↵
lazy PolymorphicFunctionObject for unwrapping a boost::reference_wrapper<> in an xpressive se↵
mantic action.
    template<typename T, typename A> unspecified as(A const &);
    template<typename T, typename A> unspecified static_cast_(A const &);
    template<typename T, typename A> unspecified dynamic_cast_(A const &);
    template<typename T, typename A> unspecified const_cast_(A const &);
    template<typename T> value< T > const val(T const &);
    template<typename T> reference< T > const ref(T &);
    template<typename T> reference< T const  > const cref(T const &);
    template<typename T> unspecified check(T const &);
    template<typename... ArgBindings> unspecified let(ArgBindings const &...);
    template<typename T, typename... Args>
      unspecified construct(Args const &...);
    namespace op {
      template<typename T> struct as;
      struct at;
      struct back;
      template<typename T> struct const_cast_;
      template<typename T> struct construct;
      template<typename T> struct dynamic_cast_;
```

```
      struct first;
      struct front;
      struct insert;
      struct length;
      struct make_pair;
      struct matched;
      struct pop;
      struct pop_back;
      struct pop_front;
      struct push;
      struct push_back;
      struct push_front;
      struct second;
      template<typename T> struct static_cast_;
      struct str;
      template<typename Except> struct throw_;
      struct top;
      struct unwrap_reference;
    }
  }
}
```

Struct template as

boost::xpressive::op::as — `as<>` is a PolymorphicFunctionObject for lexically casting a parameter to a different type.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename T>
struct as {
  // types
  typedef T result_type;

  // public member functions
  template<typename Value> T operator()(Value const &) const;
};
```

Description

Template Parameters

1. ```
   typename T
   ```

   The type to which to lexically cast the parameter.

as public member functions

1. ```
   template<typename Value> T operator()(Value const & val) const;
   ```

   Parameters: `val` The value to lexically cast.
   Returns: `boost::lexical_cast<T>(val)`

Struct at

boost::xpressive::op::at — `at` is a PolymorphicFunctionObject for indexing into a sequence

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct at {
  // member classes/structs/unions
  template<typename Sig>
  struct result {
  };
  template<typename This, typename Cont, typename Idx>
  struct result<This(Cont &, Idx)> {
    // types
    typedef Cont::reference type;
  };
  template<typename This, typename Cont, typename Idx>
  struct result<This(Cont const &, Idx)> {
    // types
    typedef Cont::const_reference type;
  };
  template<typename This, typename Cont, typename Idx>
  struct result<This(Cont, Idx)> {
    // types
    typedef Cont::const_reference type;
  };

  // public member functions
  template<typename Cont, typename Idx>
    Cont::reference operator()(Cont &, Idx) const;
  template<typename Cont, typename Idx>
    Cont::const_reference operator()(Cont const &, Idx) const;
};
```

Description

`at` public member functions

1.
```
template<typename Cont, typename Idx>
  Cont::reference operator()(Cont & c, Idx idx) const;
```

Parameters:	`c`	The RandomAccessSequence to index into
	`idx`	The index
Requires:	`Cont` is a model of RandomAccessSequence	
Returns:	`c[idx]`	

2.
```
template<typename Cont, typename Idx>
  Cont::const_reference operator()(Cont const & c, Idx idx) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Struct template result

boost::xpressive::op::at::result

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename Sig>
struct result {
};
```

Struct template result<This(Cont &, Idx)>

boost::xpressive::op::at::result<This(Cont &, Idx)>

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename This, typename Cont, typename Idx>
struct result<This(Cont &, Idx)> {
  // types
  typedef Cont::reference type;
};
```

Struct template result<This(Cont const &, Idx)>

boost::xpressive::op::at::result<This(Cont const &, Idx)>

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename This, typename Cont, typename Idx>
struct result<This(Cont const &, Idx)> {
  // types
  typedef Cont::const_reference type;
};
```

Struct template result<This(Cont, Idx)>

boost::xpressive::op::at::result<This(Cont, Idx)>

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename This, typename Cont, typename Idx>
struct result<This(Cont, Idx)> {
  // types
  typedef Cont::const_reference type;
};
```

Struct back

boost::xpressive::op::back — `back` is a PolymorphicFunctionObject for fetching the back element of a container.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct back {
  // member classes/structs/unions
  template<typename Sig>
  struct result {
  };
  template<typename This, typename Sequence>
  struct result<This(Sequence)> {
    // types
    typedef remove_reference< Sequence >::type                                    ↵
                                                 sequence_type;
    typedef mpl::if_c< is_const< sequence_type >::value, typename sequence_type::const_refer↵
ence, typename sequence_type::reference >::type type;
  };

  // public member functions
  template<typename Sequence>
    result< back(Sequence &)>::type operator()(Sequence &) const;
};
```

Description

`back` public member functions

1.
```
template<typename Sequence>
  result< back(Sequence &)>::type operator()(Sequence & seq) const;
```

Parameters: `seq` The sequence from which to fetch the back.
Returns: `seq.back()`

Struct template result

boost::xpressive::op::back::result

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename Sig>
struct result {
};
```

Struct template result<This(Sequence)>

boost::xpressive::op::back::result<This(Sequence)>

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename This, typename Sequence>
struct result<This(Sequence)> {
  // types
  typedef remove_reference< Sequence >::type                                    ↵
                                                sequence_type;
  typedef mpl::if_c< is_const< sequence_type >::value, typename sequence_type::const_refer↵
ence, typename sequence_type::reference >::type type;
};
```

Struct template const_cast_

boost::xpressive::op::const_cast_ — `const_cast_<>` is a PolymorphicFunctionObject for const-casting a parameter to a cv qualification.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename T>
struct const_cast_ {
  // types
  typedef T result_type;

  // public member functions
  template<typename Value> T operator()(Value const &) const;
};
```

Description

Template Parameters

1. ```
   typename T
   ```

   The type to which to const-cast the parameter.

`const_cast_` public member functions

1. ```
   template<typename Value> T operator()(Value const & val) const;
   ```

   Parameters: `val` The value to const-cast.
   Requires: Types `T` and `Value` differ only in cv-qualification.
   Returns: `const_cast<T>(val)`

Struct template construct

boost::xpressive::op::construct — `construct<>` is a PolymorphicFunctionObject for constructing a new object.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename T>
struct construct {
  // types
  typedef T result_type;

  // public member functions
  T operator()() const;
  template<typename A0> T operator()(A0 const &) const;
  template<typename A0, typename A1>
    T operator()(A0 const &, A1 const &) const;
  template<typename A0, typename A1, typename A2>
    T operator()(A0 const &, A1 const &, A2 const &) const;
};
```

Description

Template Parameters

1. ```
   typename T
   ```

   The type of the object to construct.

`construct` public member functions

1. ```
   T operator()() const;
   ```

   This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

2. ```
   template<typename A0> T operator()(A0 const & a0) const;
   ```

   This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

3. ```
   template<typename A0, typename A1>
     T operator()(A0 const & a0, A1 const & a1) const;
   ```

   This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

4. ```
   template<typename A0, typename A1, typename A2>
     T operator()(A0 const & a0, A1 const & a1, A2 const & a2) const;
   ```

   Parameters:
   - `a0` The first argument to the constructor
   - `a1` The second argument to the constructor
   - `a2` The third argument to the constructor

   Returns: `T(a0,a1,...)`
   ```

Struct template dynamic_cast_

boost::xpressive::op::dynamic_cast_ — `dynamic_cast_<>` is a PolymorphicFunctionObject for dynamically casting a parameter to a different type.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename T>
struct dynamic_cast_ {
  // types
  typedef T result_type;

  // public member functions
  template<typename Value> T operator()(Value const &) const;
};
```

Description

Template Parameters

1. ```
   typename T
   ```

   The type to which to dynamically cast the parameter.

`dynamic_cast_` public member functions

1. ```
   template<typename Value> T operator()(Value const & val) const;
   ```

   Parameters: `val` The value to dynamically cast.
   Returns: `dynamic_cast<T>(val)`

Struct first

boost::xpressive::op::first — `first` is a PolymorphicFunctionObject for fetching the first element of a pair.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct first {
  // member classes/structs/unions
  template<typename Sig>
  struct result {
  };
  template<typename This, typename Pair>
  struct result<This(Pair)> {
    // types
    typedef remove_reference< Pair >::type::first_type type;
  };

  // public member functions
  template<typename Pair> Pair::first_type operator()(Pair const &) const;
};
```

Description

`first` public member functions

1. ```
   template<typename Pair> Pair::first_type operator()(Pair const & p) const;
   ```

   Parameters: `p` The pair from which to fetch the first element.
   Returns: `p.first`

Struct template result

boost::xpressive::op::first::result

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename Sig>
struct result {
};
```

Struct template result<This(Pair)>

boost::xpressive::op::first::result<This(Pair)>

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename This, typename Pair>
struct result<This(Pair)> {
  // types
  typedef remove_reference< Pair >::type::first_type type;
};
```

Struct front

boost::xpressive::op::front — `front` is a PolymorphicFunctionObject for fetching the front element of a container.
```

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct front {
  // member classes/structs/unions
  template<typename Sig>
  struct result {
  };
  template<typename This, typename Sequence>
  struct result<This(Sequence)> {
    // types
    typedef remove_reference< Sequence >::type                                    ↵
                                                sequence_type;
    typedef mpl::if_c< is_const< sequence_type >::value, typename sequence_type::const_refer↵
ence, typename sequence_type::reference >::type type;
  };

  // public member functions
  template<typename Sequence>
    result< front(Sequence &)>::type operator()(Sequence &) const;
};
```

Description

`front` public member functions

1.
```
template<typename Sequence>
  result< front(Sequence &)>::type operator()(Sequence & seq) const;
```

Parameters: `seq` The sequence from which to fetch the front.
Returns: `seq.front()`

Struct template result

boost::xpressive::op::front::result

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename Sig>
struct result {
};
```

Struct template result<This(Sequence)>

boost::xpressive::op::front::result<This(Sequence)>

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename This, typename Sequence>
struct result<This(Sequence)> {
  // types
  typedef remove_reference< Sequence >::type                                    ↵
                                                sequence_type;
  typedef mpl::if_c< is_const< sequence_type >::value, typename sequence_type::const_refer↵
ence, typename sequence_type::reference >::type type;
};
```

Struct insert

boost::xpressive::op::insert — `insert` is a PolymorphicFunctionObject for inserting a value or a sequence of values into a sequence container, an associative container, or a string.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct insert {
  // member classes/structs/unions
  template<typename Sig>
  struct result {
    // types
    typedef unspecified type;
  };

  // public member functions
  template<typename Cont, typename A0>
    result< insert(Cont &, A0 const &)>::type
    operator()(Cont &, A0 const &) const;
  template<typename Cont, typename A0, typename A1>
    result< insert(Cont &, A0 const &, A1 const &)>::type
    operator()(Cont &, A0 const &, A1 const &) const;
  template<typename Cont, typename A0, typename A1, typename A2>
    result< insert(Cont &, A0 const &, A1 const &, A2 const &)>::type
    operator()(Cont &, A0 const &, A1 const &, A2 const &) const;
  template<typename Cont, typename A0, typename A1, typename A2, typename A3>
    result< insert(Cont &, A0 const &, A1 const &, A2 const &, A3 const &)>::type
    operator()(Cont &, A0 const &, A1 const &, A2 const &, A3 const &) const;
};
```

Description

`insert` public member functions

1.
```
template<typename Cont, typename A0>
  result< insert(Cont &, A0 const &)>::type
  operator()(Cont & cont, A0 const & a0) const;
```

   This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

2.
```
template<typename Cont, typename A0, typename A1>
  result< insert(Cont &, A0 const &, A1 const &)>::type
  operator()(Cont & cont, A0 const & a0, A1 const & a1) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

3.
```
template<typename Cont, typename A0, typename A1, typename A2>
  result< insert(Cont &, A0 const &, A1 const &, A2 const &)>::type
  operator()(Cont & cont, A0 const & a0, A1 const & a1, A2 const & a2) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

4.
```
template<typename Cont, typename A0, typename A1, typename A2, typename A3>
  result< insert(Cont &, A0 const &, A1 const &, A2 const &, A3 const &)>::type
  operator()(Cont & cont, A0 const & a0, A1 const & a1, A2 const & a2,
             A3 const & a3) const;
```

Parameters:

`a0`	A value, iterator, or count
`a1`	A value, iterator, string, count, or character
`a2`	A value, iterator, or count
`a3`	A count
`cont`	The container into which to insert the element(s)

Returns:

- For the form `insert()(cont, a0)`, return `cont.insert(a0)`.
- For the form `insert()(cont, a0, a1)`, return `cont.insert(a0, a1)`.
- For the form `insert()(cont, a0, a1, a2)`, return `cont.insert(a0, a1, a2)`.
- For the form `insert()(cont, a0, a1, a2, a3)`, return `cont.insert(a0, a1, a2, a3)`.

Struct template result

boost::xpressive::op::insert::result

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename Sig>
struct result {
  // types
  typedef unspecified type;
};
```

Struct length

boost::xpressive::op::length — `length` is a PolymorphicFunctionObject for fetching the length of `sub_match`.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct length {
  // member classes/structs/unions
  template<typename Sig>
  struct result {
  };
  template<typename This, typename Sub>
  struct result<This(Sub)> {
    // types
    typedef remove_reference< Sub >::type::difference_type type;
  };

  // public member functions
  template<typename Sub> Sub::difference_type operator()(Sub const &) const;
};
```

Description

`length` public member functions

1. ```
   template<typename Sub> Sub::difference_type operator()(Sub const & sub) const;
   ```

   Parameters: `sub` The `sub_match` object.
   Returns: `sub.length()`

Struct template result

boost::xpressive::op::length::result

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename Sig>
struct result {
};
```

Struct template result<This(Sub)>

boost::xpressive::op::length::result<This(Sub)>
```

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename This, typename Sub>
struct result<This(Sub)> {
  // types
  typedef remove_reference< Sub >::type::difference_type type;
};
```

Struct make_pair

boost::xpressive::op::make_pair — `make_pair` is a PolymorphicFunctionObject for building a `std::pair` out of two parameters

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct make_pair {
  // member classes/structs/unions
  template<typename Sig>
  struct result {
  };
  template<typename This, typename First, typename Second>
  struct result<This(First, Second)> {
    // types
    typedef decay< First >::type                first_type;   // For exposition only.
    typedef decay< Second >::type               second_type;  // For exposition only.
    typedef std::pair< first_type, second_type > type;
  };

  // public member functions
  template<typename First, typename Second>
    std::pair< First, Second > operator()(First const &, Second const &) const;
};
```

Description

`make_pair` public member functions

1.
```
template<typename First, typename Second>
  std::pair< First, Second >
  operator()(First const & first, Second const & second) const;
```

Parameters:	`first`	The first element of the pair
	`second`	The second element of the pair
Returns:	`std::make_pair(first, second)`	

Struct template result

boost::xpressive::op::make_pair::result

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename Sig>
struct result {
};
```

Struct template result<This(First, Second)>

boost::xpressive::op::make_pair::result<This(First, Second)>

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename This, typename First, typename Second>
struct result<This(First, Second)> {
  // types
  typedef decay< First >::type                first_type;   // For exposition only.
  typedef decay< Second >::type               second_type;  // For exposition only.
  typedef std::pair< first_type, second_type > type;
};
```

Struct matched

boost::xpressive::op::matched — `matched` is a PolymorphicFunctionObject for assessing whether a `sub_match` object matched or not.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct matched {
  // types
  typedef bool result_type;

  // public member functions
  template<typename Sub> bool operator()(Sub const &) const;
};
```

Description

`matched` public member functions

1. ```
   template<typename Sub> bool operator()(Sub const & sub) const;
   ```

   Parameters: `sub` The `sub_match` object.
   Returns: `sub.matched`
   ```

Struct pop

boost::xpressive::op::pop — pop is a PolymorphicFunctionObject for popping an element from a container.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct pop {
  // types
  typedef void result_type;

  // public member functions
  template<typename Sequence> void operator()(Sequence &) const;
};
```

Description

pop public member functions

1. ```
   template<typename Sequence> void operator()(Sequence & seq) const;
   ```

   Equivalent to `seq.pop()`.

   Parameters: `seq` The sequence from which to pop.
   Returns: `void`

Struct pop_back

boost::xpressive::op::pop_back — pop_back is a PolymorphicFunctionObject for popping an element from the back of a container.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct pop_back {
  // types
  typedef void result_type;

  // public member functions
  template<typename Sequence> void operator()(Sequence &) const;
};
```

Description

pop_back public member functions

1. ```
   template<typename Sequence> void operator()(Sequence & seq) const;
   ```

   Equivalent to `seq.pop_back()`.

   Parameters: `seq` The sequence from which to pop.
   Returns: `void`

Struct pop_front

boost::xpressive::op::pop_front — `pop_front` is a PolymorphicFunctionObject for popping an element from the front of a container.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct pop_front {
  // types
  typedef void result_type;

  // public member functions
  template<typename Sequence> void operator()(Sequence &) const;
};
```

Description

`pop_front` public member functions

1. ```
   template<typename Sequence> void operator()(Sequence & seq) const;
   ```

   Equivalent to `seq.pop_front()`.

   Parameters: `seq` The sequence from which to pop.
   Returns: `void`

Struct push

boost::xpressive::op::push — `push` is a PolymorphicFunctionObject for pushing an element into a container.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct push {
  // types
  typedef void result_type;

  // public member functions
  template<typename Sequence, typename Value>
    void operator()(Sequence &, Value const &) const;
};
```

Description

`push` public member functions

1. ```
   template<typename Sequence, typename Value>
     void operator()(Sequence & seq, Value const & val) const;
   ```

   Equivalent to `seq.push(val)`.

   Parameters: `seq` The sequence into which the value should be pushed.

	`val`	The value to push into the sequence.
Returns:	`void`	

Struct push_back

boost::xpressive::op::push_back — `push_back` is a PolymorphicFunctionObject for pushing an element into the back of a container.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct push_back {
  // types
  typedef void result_type;

  // public member functions
  template<typename Sequence, typename Value>
    void operator()(Sequence &, Value const &) const;
};
```

Description

`push_back` public member functions

1.
```
template<typename Sequence, typename Value>
  void operator()(Sequence & seq, Value const & val) const;
```

Equivalent to `seq.push_back(val)`.

Parameters:	`seq`	The sequence into which the value should be pushed.
	`val`	The value to push into the sequence.
Returns:	`void`	

Struct push_front

boost::xpressive::op::push_front — `push_front` is a PolymorphicFunctionObject for pushing an element into the front of a container.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct push_front {
  // types
  typedef void result_type;

  // public member functions
  template<typename Sequence, typename Value>
    void operator()(Sequence &, Value const &) const;
};
```

Description

`push_front` public member functions

1.
```
template<typename Sequence, typename Value>
  void operator()(Sequence & seq, Value const & val) const;
```

Equivalent to `seq.push_front(val)`.

Parameters:	`seq`	The sequence into which the value should be pushed.
	`val`	The value to push into the sequence.
Returns:	`void`	

Struct second

boost::xpressive::op::second — `second` is a PolymorphicFunctionObject for fetching the second element of a pair.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct second {
  // member classes/structs/unions
  template<typename Sig>
  struct result {
  };
  template<typename This, typename Pair>
  struct result<This(Pair)> {
    // types
    typedef remove_reference< Pair >::type::second_type type;
  };

  // public member functions
  template<typename Pair> Pair::second_type operator()(Pair const &) const;
};
```

Description

second public member functions

1.
```
template<typename Pair> Pair::second_type operator()(Pair const & p) const;
```

Parameters:	`p`	The pair from which to fetch the second element.
Returns:	`p.second`	

Struct template result

boost::xpressive::op::second::result

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename Sig>
struct result {
};
```

Struct template result<This(Pair)>

boost::xpressive::op::second::result<This(Pair)>

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename This, typename Pair>
struct result<This(Pair)> {
  // types
  typedef remove_reference< Pair >::type::second_type type;
};
```

Struct template static_cast_

boost::xpressive::op::static_cast_ — `static_cast_<>` is a PolymorphicFunctionObject for statically casting a parameter to a different type.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename T>
struct static_cast_ {
  // types
  typedef T result_type;

  // public member functions
  template<typename Value> T operator()(Value const &) const;
};
```

Description

Template Parameters

1. ```
   typename T
   ```

   The type to which to statically cast the parameter.

`static_cast_` public member functions

1. ```
   template<typename Value> T operator()(Value const & val) const;
   ```

Parameters: `val` The value to statically cast.
Returns: `static_cast<T>(val)`

Struct str

boost::xpressive::op::str — `str` is a PolymorphicFunctionObject for turning a `sub_match` into an equivalent `std::string`.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct str {
  // member classes/structs/unions
  template<typename Sig>
  struct result {
  };
  template<typename This, typename Sub>
  struct result<This(Sub)> {
    // types
    typedef remove_reference< Sub >::type::string_type type;
  };

  // public member functions
  template<typename Sub> Sub::string_type operator()(Sub const &) const;
};
```

Description

`str` public member functions

1. ```
   template<typename Sub> Sub::string_type operator()(Sub const & sub) const;
   ```

   Parameters: `sub` The `sub_match` object.
   Returns: `sub.str()`

Struct template result

boost::xpressive::op::str::result

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename Sig>
struct result {
};
```

Struct template result<This(Sub)>

boost::xpressive::op::str::result<This(Sub)>
```

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename This, typename Sub>
struct result<This(Sub)> {
  // types
  typedef remove_reference< Sub >::type::string_type type;
};
```

Struct template throw_

boost::xpressive::op::throw_ — `throw_<>` is a PolymorphicFunctionObject for throwing an exception.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename Except>
struct throw_ {
  // types
  typedef void result_type;

  // public member functions
  void operator()() const;
  template<typename A0> void operator()(A0 const &) const;
  template<typename A0, typename A1>
    void operator()(A0 const &, A1 const &) const;
  template<typename A0, typename A1, typename A2>
    void operator()(A0 const &, A1 const &, A2 const &) const;
};
```

Description

Template Parameters

1. ```
   typename Except
   ```

   The type of the object to throw.

`throw_` public member functions

1. ```
   void operator()() const;
   ```

   This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

2. ```
   template<typename A0> void operator()(A0 const & a0) const;
   ```

   This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
   ```

3.
```
template<typename A0, typename A1>
  void operator()(A0 const & a0, A1 const & a1) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

4.
```
template<typename A0, typename A1, typename A2>
  void operator()(A0 const & a0, A1 const & a1, A2 const & a2) const;
```

Note

This function makes use of the `BOOST_THROW_EXCEPTION` macro to actually throw the exception. See the documentation for the Boost.Exception library.

Parameters:
- `a0` The first argument to the constructor
- `a1` The second argument to the constructor
- `a2` The third argument to the constructor

Throws: <tt>Except(a0

Struct top

boost::xpressive::op::top — `top` is a PolymorphicFunctionObject for fetching the top element of a stack.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct top {
  // member classes/structs/unions
  template<typename Sig>
  struct result {
  };
  template<typename This, typename Sequence>
  struct result<This(Sequence)> {
    // types
    typedef remove_reference< Sequence >::type                                  ↵
                                                        sequence_type;
    typedef mpl::if_c< is_const< sequence_type >::value, typename se↵
quence_type::value_type const &, typename sequence_type::value_type & >::type type;
  };

  // public member functions
  template<typename Sequence>
    result< top(Sequence &)>::type operator()(Sequence &) const;
};
```

Description

`top` public member functions

1.
```
template<typename Sequence>
  result< top(Sequence &)>::type operator()(Sequence & seq) const;
```

Parameters:
- `seq` The sequence from which to fetch the top.

Returns: `seq.top()`

Struct template result

boost::xpressive::op::top::result

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename Sig>
struct result {
};
```

Struct template result<This(Sequence)>

boost::xpressive::op::top::result<This(Sequence)>

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename This, typename Sequence>
struct result<This(Sequence)> {
  // types
  typedef remove_reference< Sequence >::type                                    ↵
                                                  sequence_type;
  typedef mpl::if_c< is_const< sequence_type >::value, typename se↵
quence_type::value_type const &, typename sequence_type::value_type & >::type type;
};
```

Struct unwrap_reference

boost::xpressive::op::unwrap_reference — `unwrap_reference` is a PolymorphicFunctionObject for unwrapping a `boost::reference_wrapper<>`.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


struct unwrap_reference {
  // member classes/structs/unions
  template<typename Sig>
  struct result {
  };
  template<typename This, typename Ref>
  struct result<This(Ref &)> {
    // types
    typedef boost::unwrap_reference< Ref >::type & type;
  };
  template<typename This, typename Ref>
  struct result<This(Ref)> {
    // types
    typedef boost::unwrap_reference< Ref >::type & type;
  };

  // public member functions
  template<typename T> T & operator()(boost::reference_wrapper< T >) const;
};
```

Description

`unwrap_reference` public member functions

1. ```
   template<typename T> T & operator()(boost::reference_wrapper< T > r) const;
   ```

   Parameters: `r` The `boost::reference_wrapper<T>` to unwrap.
   Returns: `static_cast<T &>(r)`

Struct template result

boost::xpressive::op::unwrap_reference::result

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename Sig>
struct result {
};
```

Struct template result<This(Ref &)>

boost::xpressive::op::unwrap_reference::result<This(Ref &)>
```

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename This, typename Ref>
struct result<This(Ref &)> {
  // types
  typedef boost::unwrap_reference< Ref >::type & type;
};
```

Struct template result<This(Ref)>

boost::xpressive::op::unwrap_reference::result<This(Ref)>

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename This, typename Ref>
struct result<This(Ref)> {
  // types
  typedef boost::unwrap_reference< Ref >::type & type;
};
```

Struct template function

boost::xpressive::function — A unary metafunction that turns an ordinary function object type into the type of a deferred function object for use in xpressive semantic actions.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename PolymorphicFunctionObject>
struct function {
  // types
  typedef proto::terminal< PolymorphicFunctionObject >::type type;
};
```

Description

Use `xpressive::function<>` to turn an ordinary polymorphic function object type into a type that can be used to declare an object for use in xpressive semantic actions.

For example, the global object `xpressive::push_back` can be used to create deferred actions that have the effect of pushing a value into a container. It is defined with `xpressive::function<>` as follows:

```
xpressive::function<xpressive::op::push_back>::type const push_back = {};
```

where `op::push_back` is an ordinary function object that pushes its second argument into its first. Thus defined, `xpressive::push_back` can be used in semantic actions as follows:

```
namespace xp = boost::xpressive;
using xp::_;
std::list<int> result;
std::string str("1 23 456 7890");
xp::sregex rx = (+_d)[ xp::push_back(xp::ref(result), xp::as<int>(_) ]
    >> *(' ' >> (+_d)[ xp::push_back(xp::ref(result), xp::as<int>(_) ) ]);
```

Struct template local

boost::xpressive::local — `local<>` is a lazy wrapper for a reference to a value that is stored within the local itself. It is for use within xpressive semantic actions.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename T>
struct local : public proto::terminal::type< reference_wrapper< T > > {
  // construct/copy/destruct
  local();
  explicit local(T const &);

  // public member functions
  T & get();
  T const & get() const;
};
```

Description

Below is an example of how to use `local<>` in semantic actions.

```
using namespace boost::xpressive;
local<int> i(0);
std::string str("1!2!3?");
// count the exciting digits, but not the
// questionable ones.
sregex rex = +( _d [ ++i ] >> '!' );
regex_search(str, rex);
assert( i.get() == 2 );
```

Note

As the name "local" suggests, `local<>` objects and the regexes that refer to them should never leave the local scope. The value stored within the local object will be destroyed at the end of the `local<>'s` lifetime, and any regex objects still holding the `local<>` will be left with a dangling reference.

Template Parameters

1. `typename T`

   The type of the local variable.

`local` public construct/copy/destruct

1. ```
   local();
   ```

   Store a default-constructed value of type `T`.

2. ```
   explicit local(T const & t);
   ```

   Store a default-constructed value of type `T`.

   Parameters: `t` The initial value.

`local` public member functions

1. ```
   T & get();
   ```

   Fetch the wrapped value.

2. ```
   T const & get() const;
   ```

   This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Struct template placeholder

boost::xpressive::placeholder — For defining a placeholder to stand in for a variable a semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename T, int I = 0>
struct placeholder {
  // construct/copy/destruct
  unspecified operator=(T &) const;
  unspecified operator=(T const &) const;
};
```

Description

Use `placeholder<>` to define a placeholder for use in semantic actions to stand in for real objects. The use of placeholders allows regular expressions with actions to be defined once and reused in many contexts to read and write from objects which were not available when the regex was defined.

You can use `placeholder<>` by creating an object of type `placeholder<T>` and using that object in a semantic action exactly as you intend an object of type `T` to be used.

```
placeholder<int> _i;
placeholder<double> _d;

sregex rex = ( some >> regex >> here )
    [ ++_i, _d *= _d ];
```

Then, when doing a pattern match with either `regex_search()`, `regex_match()` or `regex_replace()`, pass a `match_results<>` object that contains bindings for the placeholders used in the regex object's semantic actions. You can create the bindings by calling `match_results::let` as follows:

```
int i = 0;
double d = 3.14;

smatch what;
what.let(_i = i)
    .let(_d = d);

if(regex_match("some string", rex, what))
   // i and d mutated here
```

If a semantic action executes that contains an unbound placeholder, a exception of type `regex_error` is thrown.

See the discussion for `xpressive::let()` and the "Referring to Non-Local Variables" section in the Users' Guide for more information.

Example:

```
// Define a placeholder for a map object:
placeholder<std::map<std::string, int> > _map;

// Match a word and an integer, separated by =>,
// and then stuff the result into a std::map<>
sregex pair = ( (s1= +_w) >> "=>" >> (s2= +_d) )
    [ _map[s1] = as<int>(s2) ];

// Match one or more word/integer pairs, separated
// by whitespace.
sregex rx = pair >> *(+_s >> pair);

// The string to parse
std::string str("aaa=>1 bbb=>23 ccc=>456");

// Here is the actual map to fill in:
std::map<std::string, int> result;

// Bind the _map placeholder to the actual map
smatch what;
what.let( _map = result );

// Execute the match and fill in result map
if(regex_match(str, what, rx))
{
    std::cout << result["aaa"] << '\n';
    std::cout << result["bbb"] << '\n';
    std::cout << result["ccc"] << '\n';
}
```

Template Parameters

1. ```
   typename T
   ```

   The type of the object for which this placeholder stands in.

2. ```
   int I = 0
   ```

An optional identifier that can be used to distinguish this placeholder from others that may be used in the same semantic action that happen to have the same type.

`placeholder` public construct/copy/destruct

1.
```
unspecified operator=(T & t) const;
```

Parameters: `t` The object to associate with this placeholder
Returns: An object of unspecified type that records the association of `t` with `*this`.

2.
```
unspecified operator=(T const & t) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Struct template reference

boost::xpressive::reference — `reference<>` is a lazy wrapper for a reference that can be used in xpressive semantic actions.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename T>
struct reference : public proto::extends< proto::terminal< reference_wrapper< T > >::type, ref↵
erence< T > >
{
  // construct/copy/destruct
  explicit reference(T &);

  // public member functions
  T & get() const;
};
```

Description

Here is an example of how to use `reference<>` to create a lazy reference to an existing object so it can be read and written in an xpressive semantic action.

```
using namespace boost::xpressive;
std::map<std::string, int> result;
reference<std::map<std::string, int> > result_ref(result);

// Match a word and an integer, separated by =>,
// and then stuff the result into a std::map<>
sregex pair = ( (s1= +_w) >> "=>" >> (s2= +_d) )
    [ result_ref[s1] = as<int>(s2) ];
```

Template Parameters

1.
```
typename T
```

The type of the referent.

`reference` public construct/copy/destruct

1.
```
explicit reference(T & t);
```

Store a reference to `t`.

Parameters: `t` Reference to object

`reference` public member functions

1.
```
T & get() const;
```

Fetch the stored value.

Struct template value

boost::xpressive::value — `value<>` is a lazy wrapper for a value that can be used in xpressive semantic actions.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename T>
struct value :
  public proto::extends< proto::terminal< T >::type, value< T > >
{
  // construct/copy/destruct
  value();
  explicit value(T const &);

  // public member functions
  T & get();
  T const & get() const;
};
```

Description

Below is an example that shows where `value<>` is useful.

```
sregex good_voodoo(boost::shared_ptr<int> pi)
{
    using namespace boost::xpressive;
    // Use val() to hold the shared_ptr by value:
    sregex rex = +( _d [ ++*val(pi) ] >> '!' );
    // OK, rex holds a reference count to the integer.
    return rex;
}
```

In the above code, `xpressive::val()` is a function that returns a `value<>` object. Had `val()` not been used here, the operation `++*pi` would have been evaluated eagerly once, instead of lazily when the regex match happens.

Template Parameters

1.
```
typename T
```

The type of the value to store.

`value` public construct/copy/destruct

1. ```
   value();
   ```

   Store a default-constructed `T`.

2. ```
   explicit value(T const & t);
   ```

   Store a copy of `t`.

   Parameters: `t` The initial value.

`value` public member functions

1. ```
   T & get();
   ```

   This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

2. ```
   T const & get() const;
   ```

   Fetch the stored value.

Global at

boost::xpressive::at — `at` is a lazy PolymorphicFunctionObject for indexing into a sequence in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::at >::type const at;
```

Global push

boost::xpressive::push — `push` is a lazy PolymorphicFunctionObject for pushing a value into a container in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::push >::type const push;
```

Global push_back

boost::xpressive::push_back — `push_back` is a lazy PolymorphicFunctionObject for pushing a value into a container in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::push_back >::type const push_back;
```

Global push_front

boost::xpressive::push_front — `push_front` is a lazy PolymorphicFunctionObject for pushing a value into a container in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::push_front >::type const push_front;
```

Global pop

boost::xpressive::pop — `pop` is a lazy PolymorphicFunctionObject for popping the top element from a sequence in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::pop >::type const pop;
```

Global pop_back

boost::xpressive::pop_back — `pop_back` is a lazy PolymorphicFunctionObject for popping the back element from a sequence in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::pop_back >::type const pop_back;
```

Global pop_front

boost::xpressive::pop_front — `pop_front` is a lazy PolymorphicFunctionObject for popping the front element from a sequence in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::pop_front >::type const pop_front;
```

Global top

boost::xpressive::top — `top` is a lazy PolymorphicFunctionObject for accessing the top element from a stack in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::top >::type const top;
```

Global back

boost::xpressive::back — `back` is a lazy PolymorphicFunctionObject for fetching the back element of a sequence in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::back >::type const back;
```

Global front

boost::xpressive::front — `front` is a lazy PolymorphicFunctionObject for fetching the front element of a sequence in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::front >::type const front;
```

Global first

boost::xpressive::first — `first` is a lazy PolymorphicFunctionObject for accessing the first element of a `std::pair<>` in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::first >::type const first;
```

Global second

boost::xpressive::second — `second` is a lazy PolymorphicFunctionObject for accessing the second element of a `std::pair<>` in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::second >::type const second;
```

Global matched

boost::xpressive::matched — `matched` is a lazy PolymorphicFunctionObject for accessing the `matched` member of a `xpressive::sub_match<>` in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::matched >::type const matched;
```

Global length

boost::xpressive::length — `length` is a lazy PolymorphicFunctionObject for computing the length of a `xpressive::sub_match<>` in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::length >::type const length;
```

Global str

boost::xpressive::str — `str` is a lazy PolymorphicFunctionObject for converting a `xpressive::sub_match<>` to a `std::basic_string<>` in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::str >::type const str;
```

Global insert

boost::xpressive::insert — `insert` is a lazy PolymorphicFunctionObject for inserting a value or a range of values into a sequence in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::insert >::type const insert;
```

Global make_pair

boost::xpressive::make_pair — `make_pair` is a lazy PolymorphicFunctionObject for making a `std::pair<>` in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::make_pair >::type const make_pair;
```

Global unwrap_reference

boost::xpressive::unwrap_reference — `unwrap_reference` is a lazy PolymorphicFunctionObject for unwrapping a `boost::reference_wrapper<>` in an xpressive semantic action.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

function< op::unwrap_reference >::type const unwrap_reference;
```

Function template as

boost::xpressive::as — `as()` is a lazy funtion for lexically casting a parameter to a different type.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename T, typename A> unspecified as(A const & a);
```

Description

Parameters:	`a` The lazy value to lexically cast.
Template Parameters:	`T` The type to which to lexically cast the parameter.
Returns:	A lazy object that, when evaluated, lexically casts its argument to the desired type.

Function template static_cast_

boost::xpressive::static_cast_ — `static_cast_` is a lazy funtion for statically casting a parameter to a different type.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename T, typename A> unspecified static_cast_(A const & a);
```

Description

Parameters:	`a` The lazy value to statically cast.
Template Parameters:	`T` The type to which to statically cast the parameter.
Returns:	A lazy object that, when evaluated, statically casts its argument to the desired type.

Function template dynamic_cast_

boost::xpressive::dynamic_cast_ — `dynamic_cast_` is a lazy funtion for dynamically casting a parameter to a different type.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename T, typename A> unspecified dynamic_cast_(A const & a);
```

Description

Parameters:	`a` The lazy value to dynamically cast.
Template Parameters:	`T` The type to which to dynamically cast the parameter.
Returns:	A lazy object that, when evaluated, dynamically casts its argument to the desired type.

Function template const_cast_

boost::xpressive::const_cast_ — `dynamic_cast_` is a lazy funtion for const-casting a parameter to a different type.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename T, typename A> unspecified const_cast_(A const & a);
```

Description

Parameters:	`a` The lazy value to const-cast.
Template Parameters:	`T` The type to which to const-cast the parameter.
Returns:	A lazy object that, when evaluated, const-casts its argument to the desired type.

Function template val

boost::xpressive::val — Helper for constructing `value<>` objects.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename T> value< T > const val(T const & t);
```

Description

Returns: `value<T>(t)`

Function template ref

boost::xpressive::ref — Helper for constructing `reference<>` objects.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename T> reference< T > const ref(T & t);
```

Description

Returns: `reference<T>(t)`

Function template cref

boost::xpressive::cref — Helper for constructing `reference<>` objects that store a reference to const.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename T> reference< T const  > const cref(T const & t);
```

Description

Returns: `reference<T const>(t)`

Function template check

boost::xpressive::check — For adding user-defined assertions to your regular expressions.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>

template<typename T> unspecified check(T const & t);
```

Description

A user-defined assertion is a kind of semantic action that evaluates a Boolean lambda and, if it evaluates to false, causes the match to fail at that location in the string. This will cause backtracking, so the match may ultimately succeed.

To use `check()` to specify a user-defined assertion in a regex, use the following syntax:

```
sregex s = (_d >> _d)[check( XXX )]; // XXX is a custom assertion
```

The assertion is evaluated with a sub_match<> object that delineates what part of the string matched the sub-expression to which the assertion was attached.

check() can be used with an ordinary predicate that takes a sub_match<> object as follows:

```
// A predicate that is true IFF a sub-match is
// either 3 or 6 characters long.
struct three_or_six
{
    bool operator()(ssub_match const &sub) const
    {
        return sub.length() == 3 || sub.length() == 6;
    }
};

// match words of 3 characters or 6 characters.
sregex rx = (bow >> +_w >> eow)[ check(three_or_six()) ] ;
```

Alternately, check() can be used to define inline custom assertions with the same syntax as is used to define semantic actions. The following code is equivalent to above:

```
// match words of 3 characters or 6 characters.
sregex rx = (bow >> +_w >> eow)[ check(length(_)==3 || length(_)==6) ] ;
```

Within a custom assertion, _ is a placeholder for the sub_match<> That delineates the part of the string matched by the sub-expression to which the custom assertion was attached.

Parameters: t The UnaryPredicate object or Boolean semantic action.

Function template let

boost::xpressive::let — For binding local variables to placeholders in semantic actions when constructing a regex_iterator or a regex_token_iterator.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename... ArgBindings> unspecified let(ArgBindings const &... args);
```

Description

xpressive::let() serves the same purpose as match_results::let(); that is, it binds a placeholder to a local value. The purpose is to allow a regex with semantic actions to be defined that refers to objects that do not yet exist. Rather than referring directly to an object, a semantic action can refer to a placeholder, and the value of the placeholder can be specified later with a *let expression*. The *let expression* created with let() is passed to the constructor of either regex_iterator or regex_token_iterator.

See the section "Referring to Non-Local Variables" in the Users' Guide for more discussion.

Example:

```
// Define a placeholder for a map object:
placeholder<std::map<std::string, int> > _map;

// Match a word and an integer, separated by =>,
// and then stuff the result into a std::map<>
sregex pair = ( (s1= +_w) >> "=>" >> (s2= +_d) )
    [ _map[s1] = as<int>(s2) ];

// The string to parse
std::string str("aaa=>1 bbb=>23 ccc=>456");

// Here is the actual map to fill in:
std::map<std::string, int> result;

// Create a regex_iterator to find all the matches
sregex_iterator it(str.begin(), str.end(), pair, let(_map=result));
sregex_iterator end;

// step through all the matches, and fill in
// the result map
while(it != end)
    ++it;

std::cout << result["aaa"] << '\n';
std::cout << result["bbb"] << '\n';
std::cout << result["ccc"] << '\n';
```

The above code displays:

```
1
23
456
```

Parameters: `args` A set of argument bindings, where each argument binding is an assignment expression, the left hand side of which must be an instance of `placeholder<X>` for some `X`, and the right hand side is an lvalue of type `X`.

Function template construct

boost::xpressive::construct — A lazy funtion for constructing objects objects of the specified type.

Synopsis

```
// In header: <boost/xpressive/regex_actions.hpp>


template<typename T, typename... Args>
  unspecified construct(Args const &... args);
```

Description

Parameters: `args` The arguments to the constructor.
Template Parameters: `T` The type of object to construct.
Returns: A lazy object that, when evaluated, returns `T(xs...)`, where `xs...` is the result of evaluating the lazy arguments `args...`.

Header <boost/xpressive/regex_algorithms.hpp>

Contains the regex_match(), regex_search() and regex_replace() algorithms.

```
namespace boost {
  namespace xpressive {
    template<typename BidiIter>
      bool regex_match(BidiIter, BidiIter, match_results< BidiIter > &,
                       basic_regex< BidiIter > const &,
                       regex_constants::match_flag_type = regex_constants::match_default);
    template<typename BidiIter>
      bool regex_match(BidiIter, BidiIter, basic_regex< BidiIter > const &,
                       regex_constants::match_flag_type = regex_constants::match_default);
    template<typename Char>
      bool regex_match(Char *, match_results< Char * > &,
                       basic_regex< Char * > const &,
                       regex_constants::match_flag_type = regex_constants::match_default);
    template<typename BidiRange, typename BidiIter>
      bool regex_match(BidiRange &, match_results< BidiIter > &,
                       basic_regex< BidiIter > const &,
                       regex_constants::match_flag_type = regex_constants::match_default,
                       unspecified = 0);
    template<typename BidiRange, typename BidiIter>
      bool regex_match(BidiRange const &, match_results< BidiIter > &,
                       basic_regex< BidiIter > const &,
                       regex_constants::match_flag_type = regex_constants::match_default,
                       unspecified = 0);
    template<typename Char>
      bool regex_match(Char *, basic_regex< Char * > const &,
                       regex_constants::match_flag_type = regex_constants::match_default);
    template<typename BidiRange, typename BidiIter>
      bool regex_match(BidiRange &, basic_regex< BidiIter > const &,
                       regex_constants::match_flag_type = regex_constants::match_default,
                       unspecified = 0);
    template<typename BidiRange, typename BidiIter>
      bool regex_match(BidiRange const &, basic_regex< BidiIter > const &,
                       regex_constants::match_flag_type = regex_constants::match_default,
                       unspecified = 0);
    template<typename BidiIter>
      bool regex_search(BidiIter, BidiIter, match_results< BidiIter > &,
                        basic_regex< BidiIter > const &,
                        regex_constants::match_flag_type = regex_constants::match_default);
    template<typename BidiIter>
      bool regex_search(BidiIter, BidiIter, basic_regex< BidiIter > const &,
                        regex_constants::match_flag_type = regex_constants::match_default);
    template<typename Char>
      bool regex_search(Char *, match_results< Char * > &,
                        basic_regex< Char * > const &,
                        regex_constants::match_flag_type = regex_constants::match_default);
    template<typename BidiRange, typename BidiIter>
      bool regex_search(BidiRange &, match_results< BidiIter > &,
                        basic_regex< BidiIter > const &,
                        regex_constants::match_flag_type = regex_constants::match_default,
                        unspecified = 0);
    template<typename BidiRange, typename BidiIter>
      bool regex_search(BidiRange const &, match_results< BidiIter > &,
                        basic_regex< BidiIter > const &,
                        regex_constants::match_flag_type = regex_constants::match_default,
                        unspecified = 0);
    template<typename Char>
      bool regex_search(Char *, basic_regex< Char * > const &,
                        regex_constants::match_flag_type = regex_constants::match_default);
```

```
    template<typename BidiRange, typename BidiIter>
      bool regex_search(BidiRange &, basic_regex< BidiIter > const &,
                        regex_constants::match_flag_type = regex_constants::match_default,
                        unspecified = 0);
    template<typename BidiRange, typename BidiIter>
      bool regex_search(BidiRange const &, basic_regex< BidiIter > const &,
                        regex_constants::match_flag_type = regex_constants::match_default,
                        unspecified = 0);
    template<typename OutIter, typename BidiIter, typename Formatter>
      OutIter regex_replace(OutIter, BidiIter, BidiIter,
                            basic_regex< BidiIter > const &,
                            Formatter const &,
                           regex_constants::match_flag_type = regex_constants::match_default,
                            unspecified = 0);
    template<typename OutIter, typename BidiIter>
      OutIter regex_replace(OutIter, BidiIter, BidiIter,
                            basic_regex< BidiIter > const &,
                            typename iterator_value< BidiIter >::type const *,
                           regex_constants::match_flag_type = regex_constants::match_default);
    template<typename BidiContainer, typename BidiIter, typename Formatter>
      BidiContainer
      regex_replace(BidiContainer &, basic_regex< BidiIter > const &,
                    Formatter const &,
                    regex_constants::match_flag_type = regex_constants::match_default,
                    unspecified = 0);
    template<typename BidiContainer, typename BidiIter, typename Formatter>
      BidiContainer
      regex_replace(BidiContainer const &, basic_regex< BidiIter > const &,
                    Formatter const &,
                    regex_constants::match_flag_type = regex_constants::match_default,
                    unspecified = 0);
    template<typename Char, typename Formatter>
      std::basic_string< typename remove_const< Char >::type >
      regex_replace(Char *, basic_regex< Char * > const &, Formatter const &,
                    regex_constants::match_flag_type = regex_constants::match_default,
                    unspecified = 0);
    template<typename BidiContainer, typename BidiIter>
      BidiContainer
      regex_replace(BidiContainer &, basic_regex< BidiIter > const &,
                    typename iterator_value< BidiIter >::type const *,
                    regex_constants::match_flag_type = regex_constants::match_default,
                    unspecified = 0);
    template<typename BidiContainer, typename BidiIter>
      BidiContainer
      regex_replace(BidiContainer const &, basic_regex< BidiIter > const &,
                    typename iterator_value< BidiIter >::type const *,
                    regex_constants::match_flag_type = regex_constants::match_default,
                    unspecified = 0);
    template<typename Char>
      std::basic_string< typename remove_const< Char >::type >
      regex_replace(Char *, basic_regex< Char * > const &,
                    typename add_const< Char >::type *,
                    regex_constants::match_flag_type = regex_constants::match_default);
  }
}
```

Function regex_match

boost::xpressive::regex_match — See if a regex matches a sequence from beginning to end.

Synopsis

```
// In header: <boost/xpressive/regex_algorithms.hpp>


template<typename BidiIter>
  bool regex_match(BidiIter begin, BidiIter end,
                   match_results< BidiIter > & what,
                   basic_regex< BidiIter > const & re,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
template<typename BidiIter>
  bool regex_match(BidiIter begin, BidiIter end,
                   basic_regex< BidiIter > const & re,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
template<typename Char>
  bool regex_match(Char * begin, match_results< Char * > & what,
                   basic_regex< Char * > const & re,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
template<typename BidiRange, typename BidiIter>
  bool regex_match(BidiRange & rng, match_results< BidiIter > & what,
                   basic_regex< BidiIter > const & re,
                   regex_constants::match_flag_type flags = regex_constants::match_default,
                   unspecified = 0);
template<typename BidiRange, typename BidiIter>
  bool regex_match(BidiRange const & rng, match_results< BidiIter > & what,
                   basic_regex< BidiIter > const & re,
                   regex_constants::match_flag_type flags = regex_constants::match_default,
                   unspecified = 0);
template<typename Char>
  bool regex_match(Char * begin, basic_regex< Char * > const & re,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
template<typename BidiRange, typename BidiIter>
  bool regex_match(BidiRange & rng, basic_regex< BidiIter > const & re,
                   regex_constants::match_flag_type flags = regex_constants::match_default,
                   unspecified = 0);
template<typename BidiRange, typename BidiIter>
  bool regex_match(BidiRange const & rng, basic_regex< BidiIter > const & re,
                   regex_constants::match_flag_type flags = regex_constants::match_default,
                   unspecified = 0);
```

Description

Determines whether there is an exact match between the regular expression `re`, and all of the sequence `[begin, end)`.

Parameters:	`begin`	The beginning of the sequence.
	`end`	The end of the sequence.
	`flags`	Optional match flags, used to control how the expression is matched against the sequence. (See `match_flag_type`.)
	`re`	The regular expression object to use
	`what`	The `match_results` struct into which the sub_matches will be written
Requires:	Type `BidiIter` meets the requirements of a Bidirectional Iterator (24.1.4).	
Requires:	`[begin,end)` denotes a valid iterator range.	
Returns:	`true` if a match is found, `false` otherwise	
Throws:	regex_error on stack exhaustion	

Function regex_search

boost::xpressive::regex_search — Determines whether there is some sub-sequence within `[begin,end)` that matches the regular expression `re`.

Synopsis

```
// In header: <boost/xpressive/regex_algorithms.hpp>


template<typename BidiIter>
  bool regex_search(BidiIter begin, BidiIter end,
                    match_results< BidiIter > & what,
                    basic_regex< BidiIter > const & re,
                    regex_constants::match_flag_type flags = regex_constants::match_default);
template<typename BidiIter>
  bool regex_search(BidiIter begin, BidiIter end,
                    basic_regex< BidiIter > const & re,
                    regex_constants::match_flag_type flags = regex_constants::match_default);
template<typename Char>
  bool regex_search(Char * begin, match_results< Char * > & what,
                    basic_regex< Char * > const & re,
                    regex_constants::match_flag_type flags = regex_constants::match_default);
template<typename BidiRange, typename BidiIter>
  bool regex_search(BidiRange & rng, match_results< BidiIter > & what,
                    basic_regex< BidiIter > const & re,
                    regex_constants::match_flag_type flags = regex_constants::match_default,
                    unspecified = 0);
template<typename BidiRange, typename BidiIter>
  bool regex_search(BidiRange const & rng, match_results< BidiIter > & what,
                    basic_regex< BidiIter > const & re,
                    regex_constants::match_flag_type flags = regex_constants::match_default,
                    unspecified = 0);
template<typename Char>
  bool regex_search(Char * begin, basic_regex< Char * > const & re,
                    regex_constants::match_flag_type flags = regex_constants::match_default);
template<typename BidiRange, typename BidiIter>
  bool regex_search(BidiRange & rng, basic_regex< BidiIter > const & re,
                    regex_constants::match_flag_type flags = regex_constants::match_default,
                    unspecified = 0);
template<typename BidiRange, typename BidiIter>
  bool regex_search(BidiRange const & rng, basic_regex< BidiIter > const & re,
                    regex_constants::match_flag_type flags = regex_constants::match_default,
                    unspecified = 0);
```

Description

Determines whether there is some sub-sequence within `[begin,end)` that matches the regular expression `re`.

Parameters:	`begin`	The beginning of the sequence
	`end`	The end of the sequence
	`flags`	Optional match flags, used to control how the expression is matched against the sequence. (See `match_flag_type`.)
	`re`	The regular expression object to use
	`what`	The `match_results` struct into which the sub_matches will be written
Requires:	Type `BidiIter` meets the requirements of a Bidirectional Iterator (24.1.4).	
Requires:	`[begin,end)` denotes a valid iterator range.	
Returns:	`true` if a match is found, `false` otherwise	
Throws:	regex_error on stack exhaustion	

Function regex_replace

boost::xpressive::regex_replace — Build an output sequence given an input sequence, a regex, and a format string or a formatter object, function, or expression.

Synopsis

```
// In header: <boost/xpressive/regex_algorithms.hpp>


template<typename OutIter, typename BidiIter, typename Formatter>
  OutIter regex_replace(OutIter out, BidiIter begin, BidiIter end,
                        basic_regex< BidiIter > const & re,
                        Formatter const & format,
                       regex_constants::match_flag_type flags = regex_constants::match_default,
                        unspecified = 0);
template<typename OutIter, typename BidiIter>
  OutIter regex_replace(OutIter out, BidiIter begin, BidiIter end,
                        basic_regex< BidiIter > const & re,
                        typename iterator_value< BidiIter >::type const * format,
                       regex_constants::match_flag_type flags = regex_constants::match_default);
template<typename BidiContainer, typename BidiIter, typename Formatter>
  BidiContainer
  regex_replace(BidiContainer & str, basic_regex< BidiIter > const & re,
                Formatter const & format,
                regex_constants::match_flag_type flags = regex_constants::match_default,
                unspecified = 0);
template<typename BidiContainer, typename BidiIter, typename Formatter>
  BidiContainer
  regex_replace(BidiContainer const & str, basic_regex< BidiIter > const & re,
                Formatter const & format,
                regex_constants::match_flag_type flags = regex_constants::match_default,
                unspecified = 0);
template<typename Char, typename Formatter>
  std::basic_string< typename remove_const< Char >::type >
  regex_replace(Char * str, basic_regex< Char * > const & re,
                Formatter const & format,
                regex_constants::match_flag_type flags = regex_constants::match_default,
                unspecified = 0);
template<typename BidiContainer, typename BidiIter>
  BidiContainer
  regex_replace(BidiContainer & str, basic_regex< BidiIter > const & re,
                typename iterator_value< BidiIter >::type const * format,
                regex_constants::match_flag_type flags = regex_constants::match_default,
                unspecified = 0);
template<typename BidiContainer, typename BidiIter>
  BidiContainer
  regex_replace(BidiContainer const & str, basic_regex< BidiIter > const & re,
                typename iterator_value< BidiIter >::type const * format,
                regex_constants::match_flag_type flags = regex_constants::match_default,
                unspecified = 0);
template<typename Char>
  std::basic_string< typename remove_const< Char >::type >
  regex_replace(Char * str, basic_regex< Char * > const & re,
                typename add_const< Char >::type * format,
                regex_constants::match_flag_type flags = regex_constants::match_default);
```

Description

Constructs a `regex_iterator` object: `regex_iterator< BidiIter > i(begin, end, re, flags)`, and uses `i` to enumerate through all of the matches m of type `match_results< BidiIter >` that occur within the sequence `[begin, end)`. If no such matches are found and `!(flags & format_no_copy)` then calls `std::copy(begin, end, out)`. Otherwise, for each match found, if `!(flags & format_no_copy)` calls `std::copy(m.prefix().first, m.prefix().second, out)`, and then calls `m.format(out, format, flags)`. Finally if `!(flags & format_no_copy)` calls `std::copy(last_m.suffix().first, last_m.suffix().second, out)` where `last_m` is a copy of the last match found.

If `flags & format_first_only` is non-zero then only the first match found is replaced.

Parameters:	`begin`	The beginning of the input sequence.
	`end`	The end of the input sequence.
	`flags`	Optional match flags, used to control how the expression is matched against the sequence. (See `match_flag_type`.)
	`format`	The format string used to format the replacement sequence, or a formatter function, function object, or expression.
	`out`	An output iterator into which the output sequence is written.
	`re`	The regular expression object to use.
Requires:	Type `BidiIter` meets the requirements of a Bidirectional Iterator (24.1.4).	
Requires:	Type `OutIter` meets the requirements of an Output Iterator (24.1.2).	
Requires:	Type `Formatter` models `ForwardRange`, `Callable<match_results<BidiIter> >`, `Callable<match_results<BidiIter>, OutIter>`, or `Callable<match_results<BidiIter>, OutIter, regex_constants::match_flag_type>`; or else it is a null-terminated format string, or an expression template representing a formatter lambda expression.	
Requires:	`[begin,end)` denotes a valid iterator range.	
Returns:	The value of the output iterator after the output sequence has been written to it.	
Throws:	regex_error on stack exhaustion or invalid format string.	

Header <boost/xpressive/regex_compiler.hpp>

Contains the definition of regex_compiler, a factory for building regex objects from strings.

```
namespace boost {
  namespace xpressive {
    template<typename BidiIter, typename RegexTraits, typename CompilerTraits>
      struct regex_compiler;
  }
}
```

Struct template regex_compiler

boost::xpressive::regex_compiler — Class template regex_compiler is a factory for building basic_regex objects from a string.

Synopsis

```
// In header: <boost/xpressive/regex_compiler.hpp>

template<typename BidiIter, typename RegexTraits, typename CompilerTraits>
struct regex_compiler {
  // types
  typedef BidiIter                              iterator_type;
  typedef iterator_value< BidiIter >::type      char_type;
  typedef regex_constants::syntax_option_type   flag_type;
  typedef RegexTraits                           traits_type;
  typedef traits_type::string_type              string_type;
  typedef traits_type::locale_type              locale_type;
  typedef traits_type::char_class_type          char_class_type;

  // construct/copy/destruct
  explicit regex_compiler(RegexTraits const & = RegexTraits());

  // public member functions
  locale_type imbue(locale_type);
  locale_type getloc() const;
  template<typename InputIter>
    basic_regex< BidiIter >
    compile(InputIter, InputIter, flag_type = regex_constants::ECMAScript);
  template<typename InputRange>
    disable_if< is_pointer< InputRange >, basic_regex< BidiIter > >::type
    compile(InputRange const &, flag_type = regex_constants::ECMAScript);
  basic_regex< BidiIter >
  compile(char_type const *, flag_type = regex_constants::ECMAScript);
  basic_regex< BidiIter > compile(char_type const *, std::size_t, flag_type);
  basic_regex< BidiIter > & operator[](string_type const &);
  basic_regex< BidiIter > const & operator[](string_type const &) const;

  // private member functions
  bool is_upper_(char_type) const;
};
```

Description

Class template regex_compiler is used to construct a basic_regex object from a string. The string should contain a valid regular expression. You can imbue a regex_compiler object with a locale, after which all basic_regex objects created with that regex_compiler object will use that locale. After creating a regex_compiler object, and optionally imbueing it with a locale, you can call the compile() method to construct a basic_regex object, passing it the string representing the regular expression. You can call compile() multiple times on the same regex_compiler object. Two basic_regex objects compiled from the same string will have different regex_id's.

`regex_compiler` public construct/copy/destruct

1. ```
   explicit regex_compiler(RegexTraits const & traits = RegexTraits());
   ```

`regex_compiler` public member functions

1. ```
   locale_type imbue(locale_type loc);
   ```

   Specify the locale to be used by a `regex_compiler`.

   Parameters: `loc` The locale that this `regex_compiler` should use.
   Returns: The previous locale.

2.
```
locale_type getloc() const;
```

Get the locale used by a `regex_compiler`.

Returns: The locale used by this regex_compiler.

3.
```
template<typename InputIter>
  basic_regex< BidiIter >
  compile(InputIter begin, InputIter end,
          flag_type flags = regex_constants::ECMAScript);
```

Builds a `basic_regex` object from a range of characters.

Parameters:	`begin`	The beginning of a range of characters representing the regular expression to compile.
	`end`	The end of a range of characters representing the regular expression to compile.
	`flags`	Optional bitmask that determines how the pat string is interpreted. (See syntax_option_type.)
Requires:	InputIter is a model of the InputIterator concept.	
Requires:	[begin,end) is a valid range.	
Requires:	The range of characters specified by [begin,end) contains a valid string-based representation of a regular expression.	
Returns:	A basic_regex object corresponding to the regular expression represented by the character range.	
Throws:	regex_error when the range of characters has invalid regular expression syntax.	

4.
```
template<typename InputRange>
  disable_if< is_pointer< InputRange >, basic_regex< BidiIter > >::type
  compile(InputRange const & pat,
          flag_type flags = regex_constants::ECMAScript);
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

5.
```
basic_regex< BidiIter >
compile(char_type const * begin,
        flag_type flags = regex_constants::ECMAScript);
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

6.
```
basic_regex< BidiIter >
compile(char_type const * begin, std::size_t size, flag_type flags);
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

7.
```
basic_regex< BidiIter > & operator[](string_type const & name);
```

Return a reference to the named regular expression. If no such named regular expression exists, create a new regular expression and return a reference to it.

Parameters:	`name`	A std::string containing the name of the regular expression.
Requires:	The string is not empty.	
Throws:	bad_alloc on allocation failure.	

8.
```
basic_regex< BidiIter > const & operator[](string_type const & name) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

`regex_compiler` private member functions

1.
```
bool is_upper_(char_type ch) const;
```

Header <boost/xpressive/regex_constants.hpp>

Contains definitions for the syntax_option_type, match_flag_type and error_type enumerations.

```
namespace boost {
  namespace xpressive {
    namespace regex_constants {
      enum syntax_option_type;
      enum match_flag_type;
      enum error_type;
    }
  }
}
```

Type syntax_option_type

boost::xpressive::regex_constants::syntax_option_type

Synopsis

```
// In header: <boost/xpressive/regex_constants.hpp>


enum syntax_option_type { ECMAScript = = 0, icase = = 1 << 1,
                          nosubs = = 1 << 2, optimize = = 1 << 3,
                          collate = = 1 << 4, single_line = = 1 << 10,
                          not_dot_null = = 1 << 11,
                          not_dot_newline = = 1 << 12,
                          ignore_white_space = = 1 << 13 };
```

Description

Flags used to customize the regex syntax

`ECMAScript`	Specifies that the grammar recognized by the regular expression engine uses its normal semantics: that is the same as that given in the ECMA-262, ECMAScript Language Specification, Chapter 15 part 10, RegExp (Regular Expression) Objects (FWD.1).
`icase`	Specifies that matching of regular expressions against a character container sequence shall be performed without regard to case.
`nosubs`	Specifies that when a regular expression is matched against a character container sequence, then no sub-expression matches are to be stored in the supplied match_results structure.
`optimize`	Specifies that the regular expression engine should pay more attention to the speed with which regular expressions are matched, and less to the speed with which regular expression objects are constructed. Otherwise it has no detectable effect on the program output.
`collate`	Specifies that character ranges of the form "[a-b]" should be locale sensitive.
`single_line`	Specifies that the ^ and $ metacharacters DO NOT match at internal line breaks. Note that this is the opposite of the perl default. It is the inverse of perl's /m (multi-line) modifier.
`not_dot_null`	Specifies that the . metacharacter does not match the null character \0.

`not_dot_newline`	Specifies that the . metacharacter does not match the newline character \n.
`ignore_white_space`	Specifies that non-escaped white-space is not significant.

Type match_flag_type

boost::xpressive::regex_constants::match_flag_type

Synopsis

```
// In header: <boost/xpressive/regex_constants.hpp>


enum match_flag_type { match_default = = 0, match_not_bol = = 1 << 1,
                       match_not_eol = = 1 << 2, match_not_bow = = 1 << 3,
                       match_not_eow = = 1 << 4, match_any = = 1 << 7,
                       match_not_null = = 1 << 8,
                       match_continuous = = 1 << 10,
                       match_partial = = 1 << 11,
                       match_prev_avail = = 1 << 12, format_default = = 0,
                       format_sed = = 1 << 13, format_perl = = 1 << 14,
                       format_no_copy = = 1 << 15,
                       format_first_only = = 1 << 16,
                       format_literal = = 1 << 17, format_all = = 1 << 18 };
```

Description

Flags used to customize the behavior of the regex algorithms

`match_default`	Specifies that matching of regular expressions proceeds without any modification of the normal rules used in ECMA-262, ECMAScript Language Specification, Chapter 15 part 10, RegExp (Regular Expression) Objects (FWD.1)
`match_not_bol`	Specifies that the expression "^" should not be matched against the sub-sequence [first,first).
`match_not_eol`	Specifies that the expression "\$" should not be matched against the sub-sequence [last,last).
`match_not_bow`	Specifies that the expression "\\b" should not be matched against the sub-sequence [first,first).
`match_not_eow`	Specifies that the expression "\\b" should not be matched against the sub-sequence [last,last).
`match_any`	Specifies that if more than one match is possible then any match is an acceptable result.
`match_not_null`	Specifies that the expression can not be matched against an empty sequence.
`match_continuous`	Specifies that the expression must match a sub-sequence that begins at first.
`match_partial`	Specifies that if no match can be found, then it is acceptable to return a match [from, last) where from != last, if there exists some sequence of characters [from,to) of which [from,last) is a prefix, and which would result in a full match.
`match_prev_avail`	Specifies that <ndash></ndash> first is a valid iterator position, when this flag is set then the flags match_not_bol and match_not_bow are ignored by the regular expression algorithms (RE.7) and iterators (RE.8).
`format_default`	Specifies that when a regular expression match is to be replaced by a new string, that the new string is constructed using the rules used by the ECMAScript replace function in ECMA-262, ECMAScript Language Specification, Chapter 15 part 5.4.11 String.prototype.replace. (FWD.1). In addition during search and replace operations then all non-overlapping occurrences of the regular expression are located and replaced, and sections of the input that did not match the expression, are copied unchanged to the output string.
`format_sed`	Specifies that when a regular expression match is to be replaced by a new string, that the new string is constructed using the rules used by the Unix sed utility in IEEE Std 1003.1-2001, Portable Operating SystemInterface (POSIX), Shells and Utilities.
`format_perl`	Specifies that when a regular expression match is to be replaced by a new string, that the new string is constructed using an implementation defined superset of the rules used by the ECMAScript replace

	function in ECMA-262, ECMAScript Language Specification, Chapter 15 part 5.4.11 String.prototype.replace (FWD.1).
`format_no_copy`	When specified during a search and replace operation, then sections of the character container sequence being searched that do match the regular expression, are not copied to the output string.
`format_first_only`	When specified during a search and replace operation, then only the first occurrence of the regular expression is replaced.
`format_literal`	Treat the format string as a literal.
`format_all`	Specifies that all syntax extensions are enabled, including conditional (?ddexpression1:expression2) replacements.

Type error_type

boost::xpressive::regex_constants::error_type

Synopsis

```
// In header: <boost/xpressive/regex_constants.hpp>


enum error_type { error_collate, error_ctype, error_escape, error_subreg,
                  error_brack, error_paren, error_brace, error_badbrace,
                  error_range, error_space, error_badrepeat, error_complexity,
                  error_stack, error_badref, error_badmark,
                  error_badlookbehind, error_badrule, error_badarg,
                  error_badattr, error_internal };
```

Description

Error codes used by the regex_error type

`error_collate`	The expression contained an invalid collating element name.
`error_ctype`	The expression contained an invalid character class name.
`error_escape`	The expression contained an invalid escaped character, or a trailing escape.
`error_subreg`	The expression contained an invalid back-reference.
`error_brack`	The expression contained mismatched [and].
`error_paren`	The expression contained mismatched (and).
`error_brace`	The expression contained mismatched { and }.
`error_badbrace`	The expression contained an invalid range in a {} expression.
`error_range`	The expression contained an invalid character range, for example [b-a].
`error_space`	There was insufficient memory to convert the expression into a finite state machine.
`error_badrepeat`	One of *?+{ was not preceded by a valid regular expression.
`error_complexity`	The complexity of an attempted match against a regular expression exceeded a pre-set level.
`error_stack`	There was insufficient memory to determine whether the regular expression could match the specified character sequence.
`error_badref`	An nested regex is uninitialized.
`error_badmark`	An invalid use of a named capture.
`error_badlookbehind`	An attempt to create a variable-width look-behind assertion was detected.
`error_badrule`	An invalid use of a rule was detected.
`error_badarg`	An argument to an action was unbound.
`error_badattr`	Tried to read from an uninitialized attribute.
`error_internal`	An internal error has occurred.

Header <boost/xpressive/regex_error.hpp>

Contains the definition of the regex_error exception class.

```
BOOST_XPR_ENSURE_(pred, code, msg)
```

```
namespace boost {
  namespace xpressive {
    struct regex_error;
  }
}
```

Struct regex_error

boost::xpressive::regex_error — The class regex_error defines the type of objects thrown as exceptions to report errors during the conversion from a string representing a regular expression to a finite state machine.

Synopsis

```
// In header: <boost/xpressive/regex_error.hpp>


struct regex_error : public std::runtime_error, public exception {
  // construct/copy/destruct
  explicit regex_error(regex_constants::error_type, char const * = "");
  ~regex_error();

  // public member functions
  regex_constants::error_type code() const;
};
```

Description

`regex_error` public construct/copy/destruct

1. ```
   explicit regex_error(regex_constants::error_type code, char const * str = "");
   ```

   Constructs an object of class `regex_error`.
   Parameters: `code` The error_type this `regex_error` represents.
   `str` The message string of this `regex_error`.
   Postconditions: code() == code

2. ```
   ~regex_error();
   ```

   Destructor for class `regex_error`
   Throws: Will not throw.

`regex_error` public member functions

1. ```
   regex_constants::error_type code() const;
   ```

   Accessor for the error_type value
   Returns: the error_type code passed to the constructor
   Throws: Will not throw.
   ```

Macro BOOST_XPR_ENSURE_

BOOST_XPR_ENSURE_

Synopsis

```
// In header: <boost/xpressive/regex_error.hpp>

BOOST_XPR_ENSURE_(pred, code, msg)
```

Header <boost/xpressive/regex_iterator.hpp>

Contains the definition of the regex_iterator type, an STL-compatible iterator for stepping through all the matches in a sequence.

```
namespace boost {
  namespace xpressive {
    template<typename BidiIter> struct regex_iterator;
  }
}
```

Struct template regex_iterator

boost::xpressive::regex_iterator

Synopsis

```
// In header: <boost/xpressive/regex_iterator.hpp>

template<typename BidiIter>
struct regex_iterator {
  // types
  typedef basic_regex< BidiIter >               regex_type;
  typedef match_results< BidiIter >             value_type;
  typedef iterator_difference< BidiIter >::type difference_type;
  typedef value_type const *                    pointer;
  typedef value_type const &                    reference;
  typedef std::forward_iterator_tag             iterator_category;

  // construct/copy/destruct
  regex_iterator();
  regex_iterator(BidiIter, BidiIter, basic_regex< BidiIter > const &,
                 regex_constants::match_flag_type = regex_constants::match_default);
  template<typename LetExpr>
    regex_iterator(BidiIter, BidiIter, basic_regex< BidiIter > const &,
                   unspecified,
                   regex_constants::match_flag_type = regex_constants::match_default);
  regex_iterator(regex_iterator< BidiIter > const &);
  regex_iterator< BidiIter > & operator=(regex_iterator< BidiIter > const &);

  // public member functions
  value_type const & operator*() const;
  value_type const * operator->() const;
  regex_iterator< BidiIter > & operator++();
  regex_iterator< BidiIter > operator++(int);
};
```

Description

`regex_iterator` public construct/copy/destruct

1.
```
regex_iterator();
```

2.
```
regex_iterator(BidiIter begin, BidiIter end,
               basic_regex< BidiIter > const & rex,
               regex_constants::match_flag_type flags = regex_constants::match_default);
```

3.
```
template<typename LetExpr>
  regex_iterator(BidiIter begin, BidiIter end,
                 basic_regex< BidiIter > const & rex, unspecified args,
                 regex_constants::match_flag_type flags = regex_constants::match_default);
```

4.
```
regex_iterator(regex_iterator< BidiIter > const & that);
```

5.
```
regex_iterator< BidiIter > &
operator=(regex_iterator< BidiIter > const & that);
```

`regex_iterator` public member functions

1.
```
value_type const & operator*() const;
```

2.
```
value_type const * operator->() const;
```

3.
```
regex_iterator< BidiIter > & operator++();
```

If what.prefix().first != what[0].second and if the element match_prev_avail is not set in flags then sets it. Then behaves as if by calling regex_search(what[0].second, end, what, *pre, flags), with the following variation: in the event that the previous match found was of zero length (what[0].length() == 0) then attempts to find a non-zero length match starting at what[0].second, only if that fails and provided what[0].second != suffix().second does it look for a (possibly zero length) match starting from what[0].second + 1. If no further match is found then sets *this equal to the end of sequence iterator.

Postconditions: (*this)->size() == pre->mark_count() + 1
Postconditions: (*this)->empty() == false
Postconditions: (*this)->prefix().first == An iterator denoting the end point of the previous match found
Postconditions: (*this)->prefix().last == (**this)[0].first
Postconditions: (*this)->prefix().matched == (*this)->prefix().first != (*this)->prefix().second
Postconditions: (*this)->suffix().first == (**this)[0].second
Postconditions: (*this)->suffix().last == end
Postconditions: (*this)->suffix().matched == (*this)->suffix().first != (*this)->suffix().second
Postconditions: (**this)[0].first == The starting iterator for this match.
Postconditions: (**this)[0].second == The ending iterator for this match.
Postconditions: (**this)[0].matched == true if a full match was found, and false if it was a partial match (found as a result of the match_partial flag being set).
Postconditions: (**this)[n].first == For all integers n < (*this)->size(), the start of the sequence that matched sub-expression n. Alternatively, if sub-expression n did not participate in the match, then end.

Postconditions:	(**this)[n].second == For all integers n < (*this)->size(), the end of the sequence that matched sub-expression n. Alternatively, if sub-expression n did not participate in the match, then end.
Postconditions:	(**this)[n].matched == For all integers n < (*this)->size(), true if sub-expression n participated in the match, false otherwise.
Postconditions:	(*this)->position() == The distance from the start of the original sequence being iterated, to the start of this match.

4.
```
regex_iterator< BidiIter > operator++(int);
```

Header <boost/xpressive/regex_primitives.hpp>

Contains the syntax elements for writing static regular expressions.

```
namespace boost {
  namespace xpressive {
    struct mark_tag;

    unsigned int const inf;    // For infinite repetition of a sub-expression.
    unspecified nil;    // Successfully matches nothing.
    unspecified alnum;    // Matches an alpha-numeric character.
    unspecified alpha;    // Matches an alphabetic character.
    unspecified blank;    // Matches a blank (horizonal white-space) character.
    unspecified cntrl;    // Matches a control character.
    unspecified digit;    // Matches a digit character.
    unspecified graph;    // Matches a graph character.
    unspecified lower;    // Matches a lower-case character.
    unspecified print;    // Matches a printable character.
    unspecified punct;    // Matches a punctuation character.
    unspecified space;    // Matches a space character.
    unspecified upper;    // Matches an upper-case character.
    unspecified xdigit;    // Matches a hexadecimal digit character.
    unspecified bos;    // Beginning of sequence assertion.
    unspecified eos;    // End of sequence assertion.
    unspecified bol;    // Beginning of line assertion.
    unspecified eol;    // End of line assertion.
    unspecified bow;    // Beginning of word assertion.
    unspecified eow;    // End of word assertion.
    unspecified _b;    // Word boundary assertion.
    unspecified _w;    // Matches a word character.
    unspecified _d;    // Matches a digit character.
    unspecified _s;    // Matches a space character.
    proto::terminal< char >::type const _n;    // Matches a literal newline character, '\n'.
    unspecified _ln;    // Matches a logical newline sequence.
    unspecified _;    // Matches any one character.
    unspecified self;    // Reference to the current regex object.
    unspecified set;    // Used to create character sets.
    mark_tag const s0;    // Sub-match placeholder, like $& in Perl.
    mark_tag const s1;    // Sub-match placeholder, like $1 in perl.
    mark_tag const s2;
    mark_tag const s3;
    mark_tag const s4;
    mark_tag const s5;
    mark_tag const s6;
    mark_tag const s7;
    mark_tag const s8;
    mark_tag const s9;
    unspecified a1;
    unspecified a2;
    unspecified a3;
    unspecified a4;
```

```
    unspecified a5;
    unspecified a6;
    unspecified a7;
    unspecified a8;
    unspecified a9;
    template<typename Expr> unspecified icase(Expr const &);
    template<typename Literal> unspecified as_xpr(Literal const &);
    template<typename BidiIter>
      proto::terminal< reference_wrapper< basic_regex< BidiIter > const  > >::type const
      by_ref(basic_regex< BidiIter > const &);
    template<typename Char> unspecified range(Char, Char);
    template<typename Expr>
      proto::result_of::make_expr< proto::tag::logical_not, proto::default_domain, Ex↵
pr const & >::type const
      optional(Expr const &);
    template<unsigned int Min, unsigned int Max, typename Expr>
      unspecified repeat(Expr const &);
    template<unsigned int Count, typename Expr2>
      unspecified repeat(Expr2 const &);
    template<typename Expr> unspecified keep(Expr const &);
    template<typename Expr> unspecified before(Expr const &);
    template<typename Expr> unspecified after(Expr const &);
    template<typename Locale> unspecified imbue(Locale const &);
    template<typename Skip> unspecified skip(Skip const &);
  }
}
```

Struct mark_tag

boost::xpressive::mark_tag — Sub-match placeholder type, used to create named captures in static regexes.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>


struct mark_tag {
  // construct/copy/destruct
  mark_tag(int);

  // private static functions
  static unspecified make_tag(int);
};
```

Description

`mark_tag` is the type of the global sub-match placeholders `s0`, `s1`, etc.. You can use the `mark_tag` type to create your own sub-match placeholders with more meaningful names. This is roughly equivalent to the "named capture" feature of dynamic regular expressions.

To create a named sub-match placeholder, initialize it with a unique integer. The integer must only be unique within the regex in which the placeholder is used. Then you can use it within static regexes to created sub-matches by assigning a sub-expression to it, or to refer back to already created sub-matches.

```
mark_tag number(1); // "number" is now equivalent to "s1"
// Match a number, followed by a space and the same number again
sregex rx = (number = +_d) >> ' ' >> number;
```

After a successful `regex_match()` or `regex_search()`, the sub-match placeholder can be used to index into the `match_results<>` object to retrieve the corresponding sub-match.

`mark_tag` public construct/copy/destruct

1.
```
mark_tag(int mark_nbr);
```

Initialize a `mark_tag` placeholder.

Parameters: `mark_nbr` An integer that uniquely identifies this `mark_tag` within the static regexes in which this `mark_tag` will be used.

Requires: `mark_nbr > 0`

`mark_tag` private static functions

1.
```
static unspecified make_tag(int mark_nbr);
```

Global inf

boost::xpressive::inf — For infinite repetition of a sub-expression.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unsigned int const inf;
```

Description

Magic value used with the repeat<>() function template to specify an unbounded repeat. Use as: repeat<17, inf>('a'). The equivalent in perl is /a{17,}/.

Global nil

boost::xpressive::nil — Successfully matches nothing.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified nil;
```

Description

Successfully matches a zero-width sequence. nil always succeeds and never consumes any characters.

Global alnum

boost::xpressive::alnum — Matches an alpha-numeric character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified alnum;
```

Description

The regex traits are used to determine which characters are alpha-numeric. To match any character that is not alpha-numeric, use ~alnum.

Note

alnum is equivalent to /[[:alnum:]]/ in perl. ~alnum is equivalent to /[[:^alnum:]]/ in perl.

Global alpha

boost::xpressive::alpha — Matches an alphabetic character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified alpha;
```

Description

The regex traits are used to determine which characters are alphabetic. To match any character that is not alphabetic, use ~alpha.

Note

alpha is equivalent to /[[:alpha:]]/ in perl. ~alpha is equivalent to /[[:^alpha:]]/ in perl.

Global blank

boost::xpressive::blank — Matches a blank (horizonal white-space) character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified blank;
```

Description

The regex traits are used to determine which characters are blank characters. To match any character that is not blank, use ~blank.

Note

blank is equivalent to /[[:blank:]]/ in perl. ~blank is equivalent to /[[:^blank:]]/ in perl.

Global cntrl

boost::xpressive::cntrl — Matches a control character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified cntrl;
```

Description

The regex traits are used to determine which characters are control characters. To match any character that is not a control character, use ~cntrl.

Note

cntrl is equivalent to /[[:cntrl:]]/ in perl. ~cntrl is equivalent to /[[:^cntrl:]]/ in perl.

Global digit

boost::xpressive::digit — Matches a digit character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified digit;
```

Description

The regex traits are used to determine which characters are digits. To match any character that is not a digit, use ~digit.

Note

digit is equivalent to /[[:digit:]]/ in perl. ~digit is equivalent to /[[:^digit:]]/ in perl.

Global graph

boost::xpressive::graph — Matches a graph character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified graph;
```

Description

The regex traits are used to determine which characters are graphable. To match any character that is not graphable, use ~graph.

Note

graph is equivalent to /[[:graph:]]/ in perl. ~graph is equivalent to /[[:^graph:]]/ in perl.

Global lower

boost::xpressive::lower — Matches a lower-case character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified lower;
```

Description

The regex traits are used to determine which characters are lower-case. To match any character that is not a lower-case character, use ~lower.

Note

lower is equivalent to /[[:lower:]]/ in perl. ~lower is equivalent to /[[:^lower:]]/ in perl.

Global print

boost::xpressive::print — Matches a printable character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified print;
```

Description

The regex traits are used to determine which characters are printable. To match any character that is not printable, use ~print.

Note

print is equivalent to /[[:print:]]/ in perl. ~print is equivalent to /[[:^print:]]/ in perl.

Global punct

boost::xpressive::punct — Matches a punctuation character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified punct;
```

Description

The regex traits are used to determine which characters are punctuation. To match any character that is not punctuation, use ~punct.

Note

punct is equivalent to /[[:punct:]]/ in perl. ~punct is equivalent to /[[:^punct:]]/ in perl.

Global space

boost::xpressive::space — Matches a space character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified space;
```

Description

The regex traits are used to determine which characters are space characters. To match any character that is not white-space, use ~space.

Note

space is equivalent to /[[:space:]]/ in perl. ~space is equivalent to /[[:^space:]]/ in perl.

Global upper

boost::xpressive::upper — Matches an upper-case character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified upper;
```

Description

The regex traits are used to determine which characters are upper-case. To match any character that is not upper-case, use ~upper.

Note

upper is equivalent to /[[:upper:]]/ in perl. ~upper is equivalent to /[[:^upper:]]/ in perl.

Global xdigit

boost::xpressive::xdigit — Matches a hexadecimal digit character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified xdigit;
```

Description

The regex traits are used to determine which characters are hex digits. To match any character that is not a hex digit, use ~xdigit.

Note

xdigit is equivalent to /[[:xdigit:]]/ in perl. ~xdigit is equivalent to /[[:^xdigit:]]/ in perl.

Global bos

boost::xpressive::bos — Beginning of sequence assertion.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified bos;
```

Description

For the character sequence [begin, end), 'bos' matches the zero-width sub-sequence [begin, begin).

Global eos

boost::xpressive::eos — End of sequence assertion.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified eos;
```

Description

For the character sequence [begin, end), 'eos' matches the zero-width sub-sequence [end, end).

Note

Unlike the perl end of sequence assertion $, 'eos' will not match at the position [end-1, end-1) if *(end-1) is '\n'. To get that behavior, use (!_n >> eos).

Global bol

boost::xpressive::bol — Beginning of line assertion.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified bol;
```

Description

'bol' matches the zero-width sub-sequence immediately following a logical newline sequence. The regex traits is used to determine what constitutes a logical newline sequence.

Global eol

boost::xpressive::eol — End of line assertion.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified eol;
```

Description

'eol' matches the zero-width sub-sequence immediately preceeding a logical newline sequence. The regex traits is used to determine what constitutes a logical newline sequence.

Global bow

boost::xpressive::bow — Beginning of word assertion.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified bow;
```

Description

'bow' matches the zero-width sub-sequence immediately following a non-word character and preceeding a word character. The regex traits are used to determine what constitutes a word character.

Global eow

boost::xpressive::eow — End of word assertion.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified eow;
```

Description

'eow' matches the zero-width sub-sequence immediately following a word character and preceeding a non-word character. The regex traits are used to determine what constitutes a word character.

Global _b

boost::xpressive::_b — Word boundary assertion.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified _b;
```

Description

'_b' matches the zero-width sub-sequence at the beginning or the end of a word. It is equivalent to (bow | eow). The regex traits are used to determine what constitutes a word character. To match a non-word boundary, use ~_b.

Note

_b is like \b in perl. ~_b is like \B in perl.

Global _w

boost::xpressive::_w — Matches a word character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified _w;
```

Description

'_w' matches a single word character. The regex traits are used to determine which characters are word characters. Use ~_w to match a character that is not a word character.

Note

_w is like \w in perl. ~_w is like \W in perl.

Global _d

boost::xpressive::_d — Matches a digit character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified _d;
```

Description

'_d' matches a single digit character. The regex traits are used to determine which characters are digits. Use ~_d to match a character that is not a digit character.

Note

_d is like \d in perl. ~_d is like \D in perl.

Global _s

boost::xpressive::_s — Matches a space character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified _s;
```

Description

'_s' matches a single space character. The regex traits are used to determine which characters are space characters. Use ~_s to match a character that is not a space character.

Note

_s is like \s in perl. ~_s is like \S in perl.

Global _n

boost::xpressive::_n — Matches a literal newline character, '\n'.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

proto::terminal< char >::type const _n;
```

Description

'_n' matches a single newline character, '\n'. Use ~_n to match a character that is not a newline.

Note

~_n is like '.' in perl without the /s modifier.

Global _ln

boost::xpressive::_ln — Matches a logical newline sequence.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified _ln;
```

Description

'_ln' matches a logical newline sequence. This can be any character in the line separator class, as determined by the regex traits, or the '\r\n' sequence. For the purpose of back-tracking, '\r\n' is treated as a unit. To match any one character that is not a logical newline, use ~_ln.

Global _

boost::xpressive::_ — Matches any one character.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified _;
```

Description

Match any character, similar to '.' in perl syntax with the /s modifier. '_' matches any one character, including the newline.

Note

To match any character except the newline, use ~_n

Global self

boost::xpressive::self — Reference to the current regex object.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified self;
```

Description

Useful when constructing recursive regular expression objects. The 'self' identifier is a short-hand for the current regex object. For instance, sregex rx = '(' >> (self | nil) >> ')'; will create a regex object that matches balanced parens such as "((()))".

Global set

boost::xpressive::set — Used to create character sets.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified set;
```

Description

There are two ways to create character sets with the 'set' identifier. The easiest is to create a comma-separated list of the characters in the set, as in (set= 'a','b','c'). This set will match 'a', 'b', or 'c'. The other way is to define the set as an argument to the set subscript operator. For instance, set['a' | range('b','c') | digit] will match an 'a', 'b', 'c' or a digit character.

To complement a set, apply the '~' operator. For instance, ~(set= 'a','b','c') will match any character that is not an 'a', 'b', or 'c'.

Sets can be composed of other, possibly complemented, sets. For instance, set[~digit | ~(set= 'a','b','c')].

Global s0

boost::xpressive::s0 — Sub-match placeholder, like $& in Perl.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

mark_tag const s0;
```

Global s1

boost::xpressive::s1 — Sub-match placeholder, like $1 in perl.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

mark_tag const s1;
```

Description

To create a sub-match, assign a sub-expression to the sub-match placeholder. For instance, (s1= _) will match any one character and remember which character was matched in the 1st sub-match. Later in the pattern, you can refer back to the sub-match. For instance, (s1= _) >> s1 will match any character, and then match the same character again.

After a successful regex_match() or regex_search(), the sub-match placeholders can be used to index into the match_results<> object to retrieve the Nth sub-match.

Global s2

boost::xpressive::s2

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

mark_tag const s2;
```

Global s3

boost::xpressive::s3

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

mark_tag const s3;
```

Global s4

boost::xpressive::s4

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

mark_tag const s4;
```

Global s5

boost::xpressive::s5

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

mark_tag const s5;
```

Global s6

boost::xpressive::s6

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

mark_tag const s6;
```

Global s7

boost::xpressive::s7

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

mark_tag const s7;
```

Global s8

boost::xpressive::s8

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

mark_tag const s8;
```

Global s9

boost::xpressive::s9

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

mark_tag const s9;
```

Global a1

boost::xpressive::a1

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified a1;
```

Global a2

boost::xpressive::a2

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified a2;
```

Global a3

boost::xpressive::a3

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified a3;
```

Global a4

boost::xpressive::a4

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified a4;
```

Global a5

boost::xpressive::a5

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified a5;
```

Global a6

boost::xpressive::a6

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified a6;
```

Global a7

boost::xpressive::a7

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified a7;
```

Global a8

boost::xpressive::a8

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified a8;
```

Global a9

boost::xpressive::a9

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

unspecified a9;
```

Function template icase

boost::xpressive::icase — Makes a sub-expression case-insensitive.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>


template<typename Expr> unspecified icase(Expr const & expr);
```

Description

Use icase() to make a sub-expression case-insensitive. For instance, "foo" >> icase(set['b'] >> "ar") will match "foo" exactly followed by "bar" irrespective of case.

Function template as_xpr

boost::xpressive::as_xpr — Makes a literal into a regular expression.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>


template<typename Literal> unspecified as_xpr(Literal const & literal);
```

Description

Use as_xpr() to turn a literal into a regular expression. For instance, "foo" >> "bar" will not compile because both operands to the right-shift operator are const char*, and no such operator exists. Use as_xpr("foo") >> "bar" instead.

You can use as_xpr() with character literals in addition to string literals. For instance, as_xpr('a') will match an 'a'. You can also complement a character literal, as with ~as_xpr('a'). This will match any one character that is not an 'a'.

Function template by_ref

boost::xpressive::by_ref — Embed a regex object by reference.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>


template<typename BidiIter>
  proto::terminal< reference_wrapper< basic_regex< BidiIter > const  > >::type const
  by_ref(basic_regex< BidiIter > const & rex);
```

Description

Parameters: `rex` The `basic_regex` object to embed by reference.

Function template range

boost::xpressive::range — Match a range of characters.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>


template<typename Char> unspecified range(Char ch_min, Char ch_max);
```

Description

Match any character in the range [ch_min, ch_max].

Parameters: `ch_max` The upper end of the range to match.
`ch_min` The lower end of the range to match.

Function template optional

boost::xpressive::optional — Make a sub-expression optional. Equivalent to !as_xpr(expr).

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>


template<typename Expr>
  proto::result_of::make_expr< proto::tag::logical_not, proto::default_domain, Ex↵
pr const & >::type const
  optional(Expr const & expr);
```

Description

Parameters: `expr` The sub-expression to make optional.

Function repeat

boost::xpressive::repeat — Repeat a sub-expression multiple times.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>


template<unsigned int Min, unsigned int Max, typename Expr>
  unspecified repeat(Expr const & expr);
template<unsigned int Count, typename Expr2>
  unspecified repeat(Expr2 const & expr2);
```

Description

There are two forms of the repeat<>() function template. To match a sub-expression N times, use repeat<N>(expr). To match a sub-expression from M to N times, use repeat<M,N>(expr).

The repeat<>() function creates a greedy quantifier. To make the quantifier non-greedy, apply the unary minus operator, as in -repeat<M,N>(expr).

Parameters: `expr` The sub-expression to repeat.

Function template keep

boost::xpressive::keep — Create an independent sub-expression.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

template<typename Expr> unspecified keep(Expr const & expr);
```

Description

Turn off back-tracking for a sub-expression. Any branches or repeats within the sub-expression will match only one way, and no other alternatives are tried.

Note

keep(expr) is equivalent to the perl (?>...) extension.

Parameters: `expr` The sub-expression to modify.

Function template before

boost::xpressive::before — Look-ahead assertion.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

template<typename Expr> unspecified before(Expr const & expr);
```

Description

before(expr) succeeds if the expr sub-expression would match at the current position in the sequence, but expr is not included in the match. For instance, before("foo") succeeds if we are before a "foo". Look-ahead assertions can be negated with the bit-compliment operator.

Note

before(expr) is equivalent to the perl (?=...) extension. ~before(expr) is a negative look-ahead assertion, equivalent to the perl (?!...) extension.

Parameters: `expr` The sub-expression to put in the look-ahead assertion.

Function template after

boost::xpressive::after — Look-behind assertion.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

template<typename Expr> unspecified after(Expr const & expr);
```

Description

after(expr) succeeds if the expr sub-expression would match at the current position minus N in the sequence, where N is the width of expr. expr is not included in the match. For instance, after("foo") succeeds if we are after a "foo". Look-behind assertions can be negated with the bit-complement operator.

Note

after(expr) is equivalent to the perl (?<=...) extension. ~after(expr) is a negative look-behind assertion, equivalent to the perl (?<!...) extension.

Parameters: `expr` The sub-expression to put in the look-ahead assertion.
Requires: expr cannot match a variable number of characters.

Function template imbue

boost::xpressive::imbue — Specify a regex traits or a std::locale.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

template<typename Locale> unspecified imbue(Locale const & loc);
```

Description

imbue() instructs the regex engine to use the specified traits or locale when matching the regex. The entire expression must use the same traits/locale. For instance, the following specifies a locale for use with a regex: std::locale loc; sregex rx = imbue(loc)(+digit);

Parameters: `loc` The std::locale or regex traits object.

Function template skip

boost::xpressive::skip — Specify which characters to skip when matching a regex.

Synopsis

```
// In header: <boost/xpressive/regex_primitives.hpp>

template<typename Skip> unspecified skip(Skip const & skip);
```

Description

`skip()` instructs the regex engine to skip certain characters when matching a regex. It is most useful for writing regexes that ignore whitespace. For instance, the following specifies a regex that skips whitespace and punctuation:

```
// A sentence is one or more words separated by whitespace
// and punctuation.
sregex word = +alpha;
sregex sentence = skip(set[_s | punct])( +word );
```

The way it works in the above example is to insert `keep(*set[_s | punct])` before each primitive within the regex. A "primitive" includes terminals like strings, character sets and nested regexes. A final `*set[_s | punct]` is added to the end of the regex. The regex `sentence` specified above is equivalent to the following:

```
sregex sentence = +( keep(*set[_s | punct]) >> word )
                     >> *set[_s | punct];
```

Note

Skipping does not affect how nested regexes are handled because they are treated atomically. String literals are also treated atomically; that is, no skipping is done within a string literal. So `skip(_s)("this that")` is not the same as `skip(_s)("this" >> as_xpr("that"))`. The first will only match when there is only one space between "this" and "that". The second will skip any and all whitespace between "this" and "that".

Parameters: `skip` A regex that specifies which characters to skip.

Header <boost/xpressive/regex_token_iterator.hpp>

Contains the definition of regex_token_iterator, and STL-compatible iterator for tokenizing a string using a regular expression.

```
namespace boost {
  namespace xpressive {
    template<typename BidiIter> struct regex_token_iterator;
  }
}
```

Struct template regex_token_iterator

boost::xpressive::regex_token_iterator

Synopsis

```
// In header: <boost/xpressive/regex_token_iterator.hpp>

template<typename BidiIter>
struct regex_token_iterator {
  // types
  typedef basic_regex< BidiIter >          regex_type;
  typedef iterator_value< BidiIter >::type char_type;
  typedef sub_match< BidiIter >            value_type;
  typedef std::ptrdiff_t                   difference_type;
  typedef value_type const *               pointer;
  typedef value_type const &               reference;
  typedef std::forward_iterator_tag        iterator_category;

  // construct/copy/destruct
  regex_token_iterator();
  regex_token_iterator(BidiIter, BidiIter, basic_regex< BidiIter > const &);
  template<typename LetExpr>
    regex_token_iterator(BidiIter, BidiIter, basic_regex< BidiIter > const &,
                         unspecified);
  template<typename Subs>
    regex_token_iterator(BidiIter, BidiIter, basic_regex< BidiIter > const &,
                         Subs const &,
                         regex_constants::match_flag_type = regex_constants::match_default);
  template<typename Subs, typename LetExpr>
    regex_token_iterator(BidiIter, BidiIter, basic_regex< BidiIter > const &,
                         Subs const &, unspecified,
                         regex_constants::match_flag_type = regex_constants::match_default);
  regex_token_iterator(regex_token_iterator< BidiIter > const &);
  regex_token_iterator< BidiIter > &
  operator=(regex_token_iterator< BidiIter > const &);

  // public member functions
  value_type const & operator*() const;
  value_type const * operator->() const;
  regex_token_iterator< BidiIter > & operator++();
  regex_token_iterator< BidiIter > operator++(int);
};
```

Description

`regex_token_iterator` public construct/copy/destruct

1.
```
regex_token_iterator();
```

Postconditions: `*this` is the end of sequence iterator.

2.
```
regex_token_iterator(BidiIter begin, BidiIter end,
                     basic_regex< BidiIter > const & rex);
```

Parameters:

`begin`	The beginning of the character range to search.
`end`	The end of the character range to search.
`rex`	The regex pattern to search for.

Requires: [begin,end) is a valid range.

3.
```
template<typename LetExpr>
  regex_token_iterator(BidiIter begin, BidiIter end,
                       basic_regex< BidiIter > const & rex, unspecified args);
```

Parameters:
- `args` A let() expression with argument bindings for semantic actions.
- `begin` The beginning of the character range to search.
- `end` The end of the character range to search.
- `rex` The regex pattern to search for.

Requires: [begin,end) is a valid range.

4.
```
template<typename Subs>
  regex_token_iterator(BidiIter begin, BidiIter end,
                       basic_regex< BidiIter > const & rex,
                       Subs const & subs,
                     regex_constants::match_flag_type flags = regex_constants::match_default);
```

Parameters:
- `begin` The beginning of the character range to search.
- `end` The end of the character range to search.
- `flags` Optional match flags, used to control how the expression is matched against the sequence. (See match_flag_type.)
- `rex` The regex pattern to search for.
- `subs` A range of integers designating sub-matches to be treated as tokens.

Requires: [begin,end) is a valid range.
Requires: `subs` is either an integer greater or equal to -1, or else an array or non-empty `std::vector<>` of such integers.

5.
```
template<typename Subs, typename LetExpr>
  regex_token_iterator(BidiIter begin, BidiIter end,
                       basic_regex< BidiIter > const & rex,
                       Subs const & subs, unspecified args,
                     regex_constants::match_flag_type flags = regex_constants::match_default);
```

Parameters:
- `args` A let() expression with argument bindings for semantic actions.
- `begin` The beginning of the character range to search.
- `end` The end of the character range to search.
- `flags` Optional match flags, used to control how the expression is matched against the sequence. (See match_flag_type.)
- `rex` The regex pattern to search for.
- `subs` A range of integers designating sub-matches to be treated as tokens.

Requires: [begin,end) is a valid range.
Requires: `subs` is either an integer greater or equal to -1, or else an array or non-empty `std::vector<>` of such integers.

6.
```
regex_token_iterator(regex_token_iterator< BidiIter > const & that);
```

Postconditions: `*this == that`

7.
```
regex_token_iterator< BidiIter > &
operator=(regex_token_iterator< BidiIter > const & that);
```

Postconditions: `*this == that`

`regex_token_iterator` public member functions

1.
```
value_type const & operator*() const;
```

2. ```
   value_type const * operator->() const;
   ```

3. ```
   regex_token_iterator< BidiIter > & operator++();
   ```

   If N == -1 then sets *this equal to the end of sequence iterator. Otherwise if N+1 < subs.size(), then increments N and sets result equal to ((subs[N] == -1) ? value_type(what.prefix().str()) : value_type(what[subs[N]].str())). Otherwise if what.prefix().first != what[0].second and if the element match_prev_avail is not set in flags then sets it. Then locates the next match as if by calling regex_search(what[0].second, end, what, *pre, flags), with the following variation: in the event that the previous match found was of zero length (what[0].length() == 0) then attempts to find a non-zero length match starting at what[0].second, only if that fails and provided what[0].second != suffix().second does it look for a (possibly zero length) match starting from what[0].second + 1. If such a match is found then sets N equal to zero, and sets result equal to ((subs[N] == -1) ? value_type(what.prefix().str()) : value_type(what[subs[N]].str())). Otherwise if no further matches were found, then let last_end be the endpoint of the last match that was found. Then if last_end != end and subs[0] == -1 sets N equal to -1 and sets result equal to value_type(last_end, end). Otherwise sets *this equal to the end of sequence iterator.

4. ```
   regex_token_iterator< BidiIter > operator++(int);
   ```

Header <boost/xpressive/regex_traits.hpp>

Includes the C regex traits or the CPP regex traits header file depending on the BOOST_XPRESSIVE_USE_C_TRAITS macro.

```
namespace boost {
  namespace xpressive {
    template<typename Traits> struct has_fold_case;
    template<typename Char, typename Impl> struct regex_traits;
    struct regex_traits_version_1_tag;
    struct regex_traits_version_2_tag;
  }
}
```

Struct template has_fold_case

boost::xpressive::has_fold_case — Trait used to denote that a traits class has the fold_case member function.

Synopsis

```
// In header: <boost/xpressive/regex_traits.hpp>

template<typename Traits>
struct has_fold_case : public is_convertible< Traits::version_tag *, regex_traits_ver↵
sion_1_case_fold_tag * >
{
};
```

Struct template regex_traits

boost::xpressive::regex_traits
```

Synopsis

```
// In header: <boost/xpressive/regex_traits.hpp>

template<typename Char, typename Impl>
struct regex_traits : public Impl {
  // types
  typedef Impl::locale_type locale_type;

  // construct/copy/destruct
  regex_traits();
  explicit regex_traits(locale_type const &);
};
```

Description

Thin wrapper around the default regex_traits implementation, either cpp_regex_traits or c_regex_traits

`regex_traits` public construct/copy/destruct

1. ```
   regex_traits();
   ```

2. ```
   explicit regex_traits(locale_type const & loc);
   ```

Struct regex_traits_version_1_tag

boost::xpressive::regex_traits_version_1_tag

Synopsis

```
// In header: <boost/xpressive/regex_traits.hpp>


struct regex_traits_version_1_tag {
};
```

Description

Tag used to denote that a traits class conforms to the version 1 traits interface.

Struct regex_traits_version_2_tag

boost::xpressive::regex_traits_version_2_tag

Synopsis

```
// In header: <boost/xpressive/regex_traits.hpp>


struct regex_traits_version_2_tag :
  public boost::xpressive::regex_traits_version_1_tag
{
};
```

Description

Tag used to denote that a traits class conforms to the version 2 traits interface.

Header <boost/xpressive/sub_match.hpp>

Contains the definition of the class template sub_match<> and associated helper functions

```
namespace boost {
  namespace xpressive {
    template<typename BidiIter> struct sub_match;
    template<typename BidiIter> BidiIter range_begin(sub_match< BidiIter > &);
    template<typename BidiIter>
      BidiIter range_begin(sub_match< BidiIter > const &);
    template<typename BidiIter> BidiIter range_end(sub_match< BidiIter > &);
    template<typename BidiIter>
      BidiIter range_end(sub_match< BidiIter > const &);
    template<typename BidiIter, typename Char, typename Traits>
      std::basic_ostream< Char, Traits > &
      operator<<(std::basic_ostream< Char, Traits > &,
                 sub_match< BidiIter > const &);
    template<typename BidiIter>
      bool operator==(sub_match< BidiIter > const & lhs,
                      sub_match< BidiIter > const & rhs);
    template<typename BidiIter>
      bool operator!=(sub_match< BidiIter > const & lhs,
                      sub_match< BidiIter > const & rhs);
    template<typename BidiIter>
      bool operator<(sub_match< BidiIter > const & lhs,
                     sub_match< BidiIter > const & rhs);
    template<typename BidiIter>
      bool operator<=(sub_match< BidiIter > const & lhs,
                      sub_match< BidiIter > const & rhs);
    template<typename BidiIter>
      bool operator>=(sub_match< BidiIter > const & lhs,
                      sub_match< BidiIter > const & rhs);
    template<typename BidiIter>
      bool operator>(sub_match< BidiIter > const & lhs,
                     sub_match< BidiIter > const & rhs);
    template<typename BidiIter>
      bool operator==(typename iterator_value< BidiIter >::type const * lhs,
                      sub_match< BidiIter > const & rhs);
    template<typename BidiIter>
      bool operator!=(typename iterator_value< BidiIter >::type const * lhs,
                      sub_match< BidiIter > const & rhs);
    template<typename BidiIter>
      bool operator<(typename iterator_value< BidiIter >::type const * lhs,
                     sub_match< BidiIter > const & rhs);
    template<typename BidiIter>
      bool operator>(typename iterator_value< BidiIter >::type const * lhs,
```

```
                sub_match< BidiIter > const & rhs);
template<typename BidiIter>
  bool operator>=(typename iterator_value< BidiIter >::type const * lhs,
                  sub_match< BidiIter > const & rhs);
template<typename BidiIter>
  bool operator<=(typename iterator_value< BidiIter >::type const * lhs,
                  sub_match< BidiIter > const & rhs);
template<typename BidiIter>
  bool operator==(sub_match< BidiIter > const & lhs,
                  typename iterator_value< BidiIter >::type const * rhs);
template<typename BidiIter>
  bool operator!=(sub_match< BidiIter > const & lhs,
                  typename iterator_value< BidiIter >::type const * rhs);
template<typename BidiIter>
  bool operator<(sub_match< BidiIter > const & lhs,
                 typename iterator_value< BidiIter >::type const * rhs);
template<typename BidiIter>
  bool operator>(sub_match< BidiIter > const & lhs,
                 typename iterator_value< BidiIter >::type const * rhs);
template<typename BidiIter>
  bool operator>=(sub_match< BidiIter > const & lhs,
                  typename iterator_value< BidiIter >::type const * rhs);
template<typename BidiIter>
  bool operator<=(sub_match< BidiIter > const & lhs,
                  typename iterator_value< BidiIter >::type const * rhs);
template<typename BidiIter>
  bool operator==(typename iterator_value< BidiIter >::type const & lhs,
                  sub_match< BidiIter > const & rhs);
template<typename BidiIter>
  bool operator!=(typename iterator_value< BidiIter >::type const & lhs,
                  sub_match< BidiIter > const & rhs);
template<typename BidiIter>
  bool operator<(typename iterator_value< BidiIter >::type const & lhs,
                 sub_match< BidiIter > const & rhs);
template<typename BidiIter>
  bool operator>(typename iterator_value< BidiIter >::type const & lhs,
                 sub_match< BidiIter > const & rhs);
template<typename BidiIter>
  bool operator>=(typename iterator_value< BidiIter >::type const & lhs,
                  sub_match< BidiIter > const & rhs);
template<typename BidiIter>
  bool operator<=(typename iterator_value< BidiIter >::type const & lhs,
                  sub_match< BidiIter > const & rhs);
template<typename BidiIter>
  bool operator==(sub_match< BidiIter > const & lhs,
                  typename iterator_value< BidiIter >::type const & rhs);
template<typename BidiIter>
  bool operator!=(sub_match< BidiIter > const & lhs,
                  typename iterator_value< BidiIter >::type const & rhs);
template<typename BidiIter>
  bool operator<(sub_match< BidiIter > const & lhs,
                 typename iterator_value< BidiIter >::type const & rhs);
template<typename BidiIter>
  bool operator>(sub_match< BidiIter > const & lhs,
                 typename iterator_value< BidiIter >::type const & rhs);
template<typename BidiIter>
  bool operator>=(sub_match< BidiIter > const & lhs,
                  typename iterator_value< BidiIter >::type const & rhs);
template<typename BidiIter>
  bool operator<=(sub_match< BidiIter > const & lhs,
                  typename iterator_value< BidiIter >::type const & rhs);
template<typename BidiIter>
  sub_match< BidiIter >::string_type
```

```
      operator+(sub_match< BidiIter > const & lhs,
                sub_match< BidiIter > const & rhs);
    template<typename BidiIter>
      sub_match< BidiIter >::string_type
      operator+(sub_match< BidiIter > const & lhs,
                typename iterator_value< BidiIter >::type const & rhs);
    template<typename BidiIter>
      sub_match< BidiIter >::string_type
      operator+(typename iterator_value< BidiIter >::type const & lhs,
                sub_match< BidiIter > const & rhs);
    template<typename BidiIter>
      sub_match< BidiIter >::string_type
      operator+(sub_match< BidiIter > const & lhs,
                typename iterator_value< BidiIter >::type const * rhs);
    template<typename BidiIter>
      sub_match< BidiIter >::string_type
      operator+(typename iterator_value< BidiIter >::type const * lhs,
                sub_match< BidiIter > const & rhs);
    template<typename BidiIter>
      sub_match< BidiIter >::string_type
      operator+(sub_match< BidiIter > const & lhs,
                typename sub_match< BidiIter >::string_type const & rhs);
    template<typename BidiIter>
      sub_match< BidiIter >::string_type
      operator+(typename sub_match< BidiIter >::string_type const & lhs,
                sub_match< BidiIter > const & rhs);
  }
}
```

Struct template sub_match

boost::xpressive::sub_match — Class template `sub_match` denotes the sequence of characters matched by a particular marked sub-expression.

Synopsis

```
// In header: <boost/xpressive/sub_match.hpp>

template<typename BidiIter>
struct sub_match : public std::pair< BidiIter, BidiIter > {
  // types
  typedef iterator_value< BidiIter >::type      value_type;
  typedef iterator_difference< BidiIter >::type difference_type;
  typedef unspecified                           string_type;
  typedef BidiIter                              iterator;

  // construct/copy/destruct
  sub_match();
  sub_match(BidiIter, BidiIter, bool = false);

  // public member functions
  string_type str() const;
  operator string_type() const;
  difference_type length() const;
  operator bool_type() const;
  bool operator!() const;
  int compare(string_type const &) const;
  int compare(sub_match const &) const;
  int compare(value_type const *) const;

  // public data members
  bool matched;  // true if this sub-match participated in the full match.
};
```

Description

When the marked sub-expression denoted by an object of type `sub_match<>` participated in a regular expression match then member `matched` evaluates to `true`, and members `first` and `second` denote the range of characters `[first,second)` which formed that match. Otherwise `matched` is `false`, and members `first` and `second` contained undefined values.

If an object of type `sub_match<>` represents sub-expression 0 - that is to say the whole match - then member `matched` is always `true`, unless a partial match was obtained as a result of the flag `match_partial` being passed to a regular expression algorithm, in which case member `matched` is `false`, and members `first` and `second` represent the character range that formed the partial match.

`sub_match` public construct/copy/destruct

1.
```
sub_match();
```

2.
```
sub_match(BidiIter first, BidiIter second, bool matched_ = false);
```

`sub_match` public member functions

1.
```
string_type str() const;
```

2.
```
operator string_type() const;
```

3.
```
difference_type length() const;
```

4.
```
operator bool_type() const;
```

5.
```
bool operator!() const;
```

6.
```
int compare(string_type const & str) const;
```

Performs a lexicographic string comparison.

Parameters: `str` the string against which to compare
Returns: the results of `(*this).str().compare(str)`

7.
```
int compare(sub_match const & sub) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

8.
```
int compare(value_type const * ptr) const;
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Function range_begin

boost::xpressive::range_begin — `range_begin()` to make `sub_match`<> a valid range

Synopsis

```
// In header: <boost/xpressive/sub_match.hpp>


template<typename BidiIter> BidiIter range_begin(sub_match< BidiIter > & sub);
template<typename BidiIter>
  BidiIter range_begin(sub_match< BidiIter > const & sub);
```

Description

Parameters: `sub` the `sub_match`<> object denoting the range
Requires: `sub.first` is not singular
Returns: `sub.first`

Function range_end

boost::xpressive::range_end — `range_end()` to make `sub_match`<> a valid range

Synopsis

```
// In header: <boost/xpressive/sub_match.hpp>


template<typename BidiIter> BidiIter range_end(sub_match< BidiIter > & sub);
template<typename BidiIter>
  BidiIter range_end(sub_match< BidiIter > const & sub);
```

Description

Parameters:	`sub`	the `sub_match<>` object denoting the range
Requires:	`sub.second` is not singular	
Returns:	`sub.second`	

Function template operator<<

boost::xpressive::operator<< — insertion operator for sending sub-matches to ostreams

Synopsis

```
// In header: <boost/xpressive/sub_match.hpp>


template<typename BidiIter, typename Char, typename Traits>
  std::basic_ostream< Char, Traits > &
  operator<<(std::basic_ostream< Char, Traits > & sout,
             sub_match< BidiIter > const & sub);
```

Description

Parameters:	`sout`	output stream.
	`sub`	`sub_match` object to be written to the stream.
Returns:	sout << sub.str()	

Header <boost/xpressive/traits/c_regex_traits.hpp>

Contains the definition of the c_regex_traits<> template, which is a wrapper for the C locale functions that can be used to customize the behavior of static and dynamic regexes.

Struct has_fold_case<c_regex_traits< char >>

boost::xpressive::has_fold_case<c_regex_traits< char >>

Synopsis

```
// In header: <boost/xpressive/traits/c_regex_traits.hpp>


struct has_fold_case<c_regex_traits< char >> : public true_ {
};
```

Header <boost/xpressive/traits/cpp_regex_traits.hpp>

Contains the definition of the cpp_regex_traits<> template, which is a wrapper for std::locale that can be used to customize the behavior of static and dynamic regexes.

Struct has_fold_case<cpp_regex_traits< char >>

boost::xpressive::has_fold_case<cpp_regex_traits< char >>

Synopsis

```
// In header: <boost/xpressive/traits/cpp_regex_traits.hpp>


struct has_fold_case<cpp_regex_traits< char >> : public true_ {
};
```

Header <boost/xpressive/traits/null_regex_traits.hpp>

Contains the definition of the null_regex_traits<> template, which is a stub regex traits implementation that can be used by static and dynamic regexes for searching non-character data.

Header <boost/xpressive/xpressive.hpp>

Includes all of xpressive including support for both static and dynamic regular expressions.

Header <boost/xpressive/xpressive_dynamic.hpp>

Includes everything you need to write and use dynamic regular expressions.

Header <boost/xpressive/xpressive_fwd.hpp>

Forward declarations for all of xpressive's public data types.

```


BOOST_PROTO_FUSION_V2
BOOST_XPRESSIVE_HAS_MS_STACK_GUARD
```

```
namespace boost {
  namespace xpressive {
    template<typename Char> struct c_regex_traits;
    template<typename RegexTraits> struct compiler_traits;
    template<typename Char> struct cpp_regex_traits;
    template<typename Elem> struct null_regex_traits;

    typedef void const * regex_id_type;
    typedef basic_regex< std::string::const_iterator > sregex;
    typedef basic_regex< char const * > cregex;
    typedef basic_regex< std::wstring::const_iterator > wsregex;
    typedef basic_regex< wchar_t const * > wcregex;
    typedef sub_match< std::string::const_iterator > ssub_match;
    typedef sub_match< char const * > csub_match;
    typedef sub_match< std::wstring::const_iterator > wssub_match;
    typedef sub_match< wchar_t const * > wcsub_match;
    typedef regex_compiler< std::string::const_iterator > sregex_compiler;
    typedef regex_compiler< char const * > cregex_compiler;
    typedef regex_compiler< std::wstring::const_iterator > wsregex_compiler;
    typedef regex_compiler< wchar_t const * > wcregex_compiler;
    typedef regex_iterator< std::string::const_iterator > sregex_iterator;
    typedef regex_iterator< char const * > cregex_iterator;
    typedef regex_iterator< std::wstring::const_iterator > wsregex_iterator;
    typedef regex_iterator< wchar_t const * > wcregex_iterator;
    typedef regex_token_iterator< std::string::const_iterator > sregex_token_iterator;
    typedef regex_token_iterator< char const * > cregex_token_iterator;
    typedef regex_token_iterator< std::wstring::const_iterator > wsregex_token_iterator;
    typedef regex_token_iterator< wchar_t const * > wcregex_token_iterator;
    typedef match_results< std::string::const_iterator > smatch;
    typedef match_results< char const * > cmatch;
    typedef match_results< std::wstring::const_iterator > wsmatch;
    typedef match_results< wchar_t const * > wcmatch;
   typedef regex_id_filter_predicate< std::string::const_iterator > sregex_id_filter_predicate;
    typedef regex_id_filter_predicate< char const * > cregex_id_filter_predicate;
   typedef regex_id_filter_predicate< std::wstring::const_iterator > wsregex_id_filter_predicate;
    typedef regex_id_filter_predicate< wchar_t const * > wcregex_id_filter_predicate;
    namespace op {
    }
  }
}
```

Struct template c_regex_traits

boost::xpressive::c_regex_traits — Encapsaulates the standard C locale functions for use by the `basic_regex<>` class template.

Synopsis

```
// In header: <boost/xpressive/xpressive_fwd.hpp>

template<typename Char>
struct c_regex_traits {
  // construct/copy/destruct
  c_regex_traits(locale_type const & = locale_type());

  // public member functions
  bool operator==(c_regex_traits< char_type > const &) const;
  bool operator!=(c_regex_traits< char_type > const &) const;
  string_type fold_case(char_type) const;
  locale_type imbue(locale_type);
  template<> char widen(char);
  template<> wchar_t widen(char);
  template<> unsigned char hash(char);
  template<> unsigned char hash(wchar_t);
  template<> int value(char, int);
  template<> int value(wchar_t, int);

  // public static functions
  static char_type widen(char);
  static unsigned char hash(char_type);
  static char_type translate(char_type);
  static char_type translate_nocase(char_type);
  static char_type tolower(char_type);
  static char_type toupper(char_type);
  static bool in_range(char_type, char_type, char_type);
  static bool in_range_nocase(char_type, char_type, char_type);
  template<typename FwdIter> static string_type transform(FwdIter, FwdIter);
  template<typename FwdIter>
    static string_type transform_primary(FwdIter, FwdIter);
  template<typename FwdIter>
    static string_type lookup_collatename(FwdIter, FwdIter);
  template<typename FwdIter>
    static char_class_type lookup_classname(FwdIter, FwdIter, bool);
  static bool isctype(char_type, char_class_type);
  static int value(char_type, int);
  static locale_type getloc();
};
```

Description

`c_regex_traits` public construct/copy/destruct

1. ```
   c_regex_traits(locale_type const & loc = locale_type());
   ```

   Initialize a `c_regex_traits` object to use the global C locale.

`c_regex_traits` public member functions

1. ```
   bool operator==(c_regex_traits< char_type > const &) const;
   ```

   Checks two `c_regex_traits` objects for equality

   Returns: true.

2. ```
   bool operator!=(c_regex_traits< char_type > const &) const;
   ```
   ```

Checks two `c_regex_traits` objects for inequality

Returns: false.

3.
```
string_type fold_case(char_type ch) const;
```

Returns a `string_type` containing all the characters that compare equal disregrarding case to the one passed in. This function can only be called if `has_fold_case<c_regex_traits<Char> >::value` is `true`.

Parameters: `ch` The source character.
Returns: `string_type` containing all chars which are equal to `ch` when disregarding case

4.
```
locale_type imbue(locale_type loc);
```

No-op

5.
```
template<> char widen(char ch);
```

6.
```
template<> wchar_t widen(char ch);
```

7.
```
template<> unsigned char hash(char ch);
```

8.
```
template<> unsigned char hash(wchar_t ch);
```

9.
```
template<> int value(char ch, int radix);
```

10.
```
template<> int value(wchar_t ch, int radix);
```

`c_regex_traits` public static functions

1.
```
static char_type widen(char ch);
```

Convert a char to a Char

Parameters: `ch` The source character.
Returns: ch if Char is char, std::btowc(ch) if Char is wchar_t.

2.
```
static unsigned char hash(char_type ch);
```

Returns a hash value for a Char in the range [0, UCHAR_MAX]

Parameters: `ch` The source character.
Returns: a value between 0 and UCHAR_MAX, inclusive.

3.
```
static char_type translate(char_type ch);
```

No-op

Parameters: `ch` The source character.
Returns: ch

4.
```
static char_type translate_nocase(char_type ch);
```

Converts a character to lower-case using the current global C locale.

Parameters: `ch` The source character.
Returns: std::tolower(ch) if Char is char, std::towlower(ch) if Char is wchar_t.

5.
```
static char_type tolower(char_type ch);
```

Converts a character to lower-case using the current global C locale.

Parameters: `ch` The source character.
Returns: std::tolower(ch) if Char is char, std::towlower(ch) if Char is wchar_t.

6.
```
static char_type toupper(char_type ch);
```

Converts a character to upper-case using the current global C locale.

Parameters: `ch` The source character.
Returns: std::toupper(ch) if Char is char, std::towupper(ch) if Char is wchar_t.

7.
```
static bool in_range(char_type first, char_type last, char_type ch);
```

Checks to see if a character is within a character range.

Parameters: `ch` The source character.
`first` The bottom of the range, inclusive.
`last` The top of the range, inclusive.
Returns: first <= ch && ch <= last.

8.
```
static bool in_range_nocase(char_type first, char_type last, char_type ch);
```

Checks to see if a character is within a character range, irregardless of case.

Note

The default implementation doesn't do proper Unicode case folding, but this is the best we can do with the standard C locale functions.

Parameters: `ch` The source character.
`first` The bottom of the range, inclusive.
`last` The top of the range, inclusive.
Returns: in_range(first, last, ch) || in_range(first, last, tolower(ch)) || in_range(first, last, toupper(ch))

9.
```
template<typename FwdIter>
  static string_type transform(FwdIter begin, FwdIter end);
```

Returns a sort key for the character sequence designated by the iterator range [F1, F2) such that if the character sequence [G1, G2) sorts before the character sequence [H1, H2) then v.transform(G1, G2) < v.transform(H1, H2).

Note

Not currently used

10.
```
template<typename FwdIter>
  static string_type transform_primary(FwdIter begin, FwdIter end);
```

Returns a sort key for the character sequence designated by the iterator range [F1, F2) such that if the character sequence [G1, G2) sorts before the character sequence [H1, H2) when character case is not considered then v.transform_primary(G1, G2) < v.transform_primary(H1, H2).

Note

Not currently used

11.
```
template<typename FwdIter>
  static string_type lookup_collatename(FwdIter begin, FwdIter end);
```

Returns a sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range [F1, F2). Returns an empty string if the character sequence is not a valid collating element.

Note

Not currently used

12.
```
template<typename FwdIter>
  static char_class_type
  lookup_classname(FwdIter begin, FwdIter end, bool icase);
```

For the character class name represented by the specified character sequence, return the corresponding bitmask representation.

Parameters:	`begin`	A forward iterator to the start of the character sequence representing the name of the character class.
	`end`	The end of the character sequence.
	`icase`	Specifies whether the returned bitmask should represent the case-insensitive version of the character class.
Returns:	A bitmask representing the character class.	

13.
```
static bool isctype(char_type ch, char_class_type mask);
```

Tests a character against a character class bitmask.

Parameters:	`ch`	The character to test.
	`mask`	The character class bitmask against which to test.
Requires:	mask is a bitmask returned by lookup_classname, or is several such masks bit-or'ed together.	
Returns:	true if the character is a member of any of the specified character classes, false otherwise.	

14.
```
static int value(char_type ch, int radix);
```

Convert a digit character into the integer it represents.

Parameters:	`ch`	The digit character.
	`radix`	The radix to use for the conversion.
Requires:	radix is one of 8, 10, or 16.	
Returns:	-1 if ch is not a digit character, the integer value of the character otherwise. If char_type is char, std::strtol is used for the conversion. If char_type is wchar_t, std::wcstol is used.	

15.
```
static locale_type getloc();
```

No-op

Struct template compiler_traits

boost::xpressive::compiler_traits

Synopsis

```
// In header: <boost/xpressive/xpressive_fwd.hpp>

template<typename RegexTraits>
struct compiler_traits {
};
```

Struct template cpp_regex_traits

boost::xpressive::cpp_regex_traits — Encapsaulates a `std::locale` for use by the `basic_regex`<> class template.

Synopsis

```
// In header: <boost/xpressive/xpressive_fwd.hpp>

template<typename Char>
struct cpp_regex_traits {
  // construct/copy/destruct
  cpp_regex_traits(locale_type const & = locale_type());

  // public member functions
  bool operator==(cpp_regex_traits< char_type > const &) const;
  bool operator!=(cpp_regex_traits< char_type > const &) const;
  char_type widen(char) const;
  char_type translate_nocase(char_type) const;
  char_type tolower(char_type) const;
  char_type toupper(char_type) const;
  string_type fold_case(char_type) const;
  bool in_range_nocase(char_type, char_type, char_type) const;
  template<typename FwdIter>
    string_type transform_primary(FwdIter, FwdIter) const;
  template<typename FwdIter>
    string_type lookup_collatename(FwdIter, FwdIter) const;
  template<typename FwdIter>
    char_class_type lookup_classname(FwdIter, FwdIter, bool) const;
  bool isctype(char_type, char_class_type) const;
  int value(char_type, int) const;
  locale_type imbue(locale_type);
  locale_type getloc() const;
  template<> unsigned char hash(unsigned char);
  template<> unsigned char hash(char);
  template<> unsigned char hash(signed char);
  template<> unsigned char hash(wchar_t);

  // public static functions
  static unsigned char hash(char_type);
  static char_type translate(char_type);
  static bool in_range(char_type, char_type, char_type);
};
```

Description

`cpp_regex_traits` public construct/copy/destruct

1. ```
   cpp_regex_traits(locale_type const & loc = locale_type());
   ```

   Initialize a `cpp_regex_traits` object to use the specified std::locale, or the global std::locale if none is specified.

`cpp_regex_traits` public member functions

1. ```
   bool operator==(cpp_regex_traits< char_type > const & that) const;
   ```

   Checks two `cpp_regex_traits` objects for equality

   Returns: this->getloc() == that.getloc().

2. ```
   bool operator!=(cpp_regex_traits< char_type > const & that) const;
   ```

   Checks two `cpp_regex_traits` objects for inequality
   ```

Returns: this->getloc() != that.getloc().

3.
```
char_type widen(char ch) const;
```

Convert a char to a Char

Parameters: `ch` The source character.
Returns: std::use_facet<std::ctype<char_type> >(this->getloc()).widen(ch).

4.
```
char_type translate_nocase(char_type ch) const;
```

Converts a character to lower-case using the internally-stored std::locale.

Parameters: `ch` The source character.
Returns: std::tolower(ch, this->getloc()).

5.
```
char_type tolower(char_type ch) const;
```

Converts a character to lower-case using the internally-stored std::locale.

Parameters: `ch` The source character.
Returns: std::tolower(ch, this->getloc()).

6.
```
char_type toupper(char_type ch) const;
```

Converts a character to upper-case using the internally-stored std::locale.

Parameters: `ch` The source character.
Returns: std::toupper(ch, this->getloc()).

7.
```
string_type fold_case(char_type ch) const;
```

Returns a `string_type` containing all the characters that compare equal disregrarding case to the one passed in. This function can only be called if `has_fold_case<cpp_regex_traits<Char> >::value` is `true`.

Parameters: `ch` The source character.
Returns: `string_type` containing all chars which are equal to `ch` when disregarding case

8.
```
bool in_range_nocase(char_type first, char_type last, char_type ch) const;
```

Checks to see if a character is within a character range, irregardless of case.

Note

The default implementation doesn't do proper Unicode case folding, but this is the best we can do with the standard ctype facet.

Parameters: `ch` The source character.
`first` The bottom of the range, inclusive.
`last` The top of the range, inclusive.
Returns: in_range(first, last, ch) || in_range(first, last, tolower(ch, this->getloc())) || in_range(first, last, toupper(ch, this->getloc()))

9.
```
template<typename FwdIter>
  string_type transform_primary(FwdIter, FwdIter) const;
```

Returns a sort key for the character sequence designated by the iterator range [F1, F2) such that if the character sequence [G1, G2) sorts before the character sequence [H1, H2) when character case is not considered then v.transform_primary(G1, G2) < v.transform_primary(H1, H2).

Note

Not currently used

10.
```
template<typename FwdIter>
  string_type lookup_collatename(FwdIter, FwdIter) const;
```

Returns a sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range [F1, F2). Returns an empty string if the character sequence is not a valid collating element.

Note

Not currently used

11.
```
template<typename FwdIter>
  char_class_type
  lookup_classname(FwdIter begin, FwdIter end, bool icase) const;
```

For the character class name represented by the specified character sequence, return the corresponding bitmask representation.

Parameters:	`begin`	A forward iterator to the start of the character sequence representing the name of the character class.
	`end`	The end of the character sequence.
	`icase`	Specifies whether the returned bitmask should represent the case-insensitive version of the character class.
Returns:	A bitmask representing the character class.	

12.
```
bool isctype(char_type ch, char_class_type mask) const;
```

Tests a character against a character class bitmask.

Parameters:	`ch`	The character to test.
	`mask`	The character class bitmask against which to test.
Requires:	mask is a bitmask returned by lookup_classname, or is several such masks bit-or'ed together.	
Returns:	true if the character is a member of any of the specified character classes, false otherwise.	

13.
```
int value(char_type ch, int radix) const;
```

Convert a digit character into the integer it represents.

Parameters:	`ch`	The digit character.
	`radix`	The radix to use for the conversion.
Requires:	radix is one of 8, 10, or 16.	

Returns: -1 if ch is not a digit character, the integer value of the character otherwise. The conversion is performed by imbueing a std::stringstream with this->getloc(); setting the radix to one of oct, hex or dec; inserting ch into the stream; and extracting an int.

14.
```
locale_type imbue(locale_type loc);
```

Imbues *this with loc

Parameters: `loc` A std::locale.
Returns: the previous std::locale used by *this.

15.
```
locale_type getloc() const;
```

Returns the current std::locale used by *this.

16.
```
template<> unsigned char hash(unsigned char ch);
```

17.
```
template<> unsigned char hash(char ch);
```

18.
```
template<> unsigned char hash(signed char ch);
```

19.
```
template<> unsigned char hash(wchar_t ch);
```

`cpp_regex_traits` public static functions

1.
```
static unsigned char hash(char_type ch);
```

Returns a hash value for a Char in the range [0, UCHAR_MAX]

Parameters: `ch` The source character.
Returns: a value between 0 and UCHAR_MAX, inclusive.

2.
```
static char_type translate(char_type ch);
```

No-op

Parameters: `ch` The source character.
Returns: ch

3.
```
static bool in_range(char_type first, char_type last, char_type ch);
```

Checks to see if a character is within a character range.

Parameters: `ch` The source character.
`first` The bottom of the range, inclusive.
`last` The top of the range, inclusive.
Returns: first <= ch && ch <= last.

Struct template null_regex_traits

boost::xpressive::null_regex_traits — stub regex_traits for non-char data

Synopsis

```
// In header: <boost/xpressive/xpressive_fwd.hpp>

template<typename Elem>
struct null_regex_traits {
  // construct/copy/destruct
  null_regex_traits(locale_type = locale_type());

  // public member functions
  bool operator==(null_regex_traits< char_type > const &) const;
  bool operator!=(null_regex_traits< char_type > const &) const;
  char_type widen(char) const;

  // public static functions
  static unsigned char hash(char_type);
  static char_type translate(char_type);
  static char_type translate_nocase(char_type);
  static bool in_range(char_type, char_type, char_type);
  static bool in_range_nocase(char_type, char_type, char_type);
  template<typename FwdIter> static string_type transform(FwdIter, FwdIter);
  template<typename FwdIter>
    static string_type transform_primary(FwdIter, FwdIter);
  template<typename FwdIter>
    static string_type lookup_collatename(FwdIter, FwdIter);
  template<typename FwdIter>
    static char_class_type lookup_classname(FwdIter, FwdIter, bool);
  static bool isctype(char_type, char_class_type);
  static int value(char_type, int);
  static locale_type imbue(locale_type);
  static locale_type getloc();
};
```

Description

`null_regex_traits` public construct/copy/destruct

1. ```
   null_regex_traits(locale_type = locale_type());
   ```

   Initialize a `null_regex_traits` object.

`null_regex_traits` public member functions

1. ```
   bool operator==(null_regex_traits< char_type > const & that) const;
   ```

   Checks two `null_regex_traits` objects for equality

   Returns: true.

2. ```
   bool operator!=(null_regex_traits< char_type > const & that) const;
   ```

   Checks two `null_regex_traits` objects for inequality

   Returns: false.
   ```

3.
```
char_type widen(char ch) const;
```

Convert a char to a Elem

Parameters: `ch` The source character.
Returns: Elem(ch).

`null_regex_traits` public static functions

1.
```
static unsigned char hash(char_type ch);
```

Returns a hash value for a Elem in the range [0, UCHAR_MAX]

Parameters: `ch` The source character.
Returns: a value between 0 and UCHAR_MAX, inclusive.

2.
```
static char_type translate(char_type ch);
```

No-op

Parameters: `ch` The source character.
Returns: ch

3.
```
static char_type translate_nocase(char_type ch);
```

No-op

Parameters: `ch` The source character.
Returns: ch

4.
```
static bool in_range(char_type first, char_type last, char_type ch);
```

Checks to see if a character is within a character range.

Parameters: `ch` The source character.
`first` The bottom of the range, inclusive.
`last` The top of the range, inclusive.
Returns: first <= ch && ch <= last.

5.
```
static bool in_range_nocase(char_type first, char_type last, char_type ch);
```

Checks to see if a character is within a character range.

Note

Since the `null_regex_traits` does not do case-folding, this function is equivalent to in_range().

Parameters: `ch` The source character.
`first` The bottom of the range, inclusive.
`last` The top of the range, inclusive.
Returns: first <= ch && ch <= last.

6.
```
template<typename FwdIter>
  static string_type transform(FwdIter begin, FwdIter end);
```

Returns a sort key for the character sequence designated by the iterator range [F1, F2) such that if the character sequence [G1, G2) sorts before the character sequence [H1, H2) then v.transform(G1, G2) < v.transform(H1, H2).

Note

Not currently used

7.
```
template<typename FwdIter>
  static string_type transform_primary(FwdIter begin, FwdIter end);
```

Returns a sort key for the character sequence designated by the iterator range [F1, F2) such that if the character sequence [G1, G2) sorts before the character sequence [H1, H2) when character case is not considered then v.transform_primary(G1, G2) < v.transform_primary(H1, H2).

Note

Not currently used

8.
```
template<typename FwdIter>
  static string_type lookup_collatename(FwdIter begin, FwdIter end);
```

Returns a sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range [F1, F2). Returns an empty string if the character sequence is not a valid collating element.

Note

Not currently used

9.
```
template<typename FwdIter>
  static char_class_type
  lookup_classname(FwdIter begin, FwdIter end, bool icase);
```

The `null_regex_traits` does not have character classifications, so lookup_classname() is unused.

Parameters:	`begin`	not used
	`end`	not used
	`icase`	not used
Returns:	static_cast<char_class_type>(0)	

10.
```
static bool isctype(char_type ch, char_class_type mask);
```

The `null_regex_traits` does not have character classifications, so isctype() is unused.

Parameters:	`ch`	not used
	`mask`	not used
Returns:	false	

11.
```
static int value(char_type ch, int radix);
```

The null_regex_traits recognizes no elements as digits, so value() is unused.

Parameters: ch not used
radix not used
Returns: -1

12.
```
static locale_type imbue(locale_type loc);
```

Not used

Parameters: loc not used
Returns: loc

13.
```
static locale_type getloc();
```

Returns locale_type().

Returns: locale_type()

Macro BOOST_PROTO_FUSION_V2

BOOST_PROTO_FUSION_V2

Synopsis

```
// In header: <boost/xpressive/xpressive_fwd.hpp>

BOOST_PROTO_FUSION_V2
```

Macro BOOST_XPRESSIVE_HAS_MS_STACK_GUARD

BOOST_XPRESSIVE_HAS_MS_STACK_GUARD

Synopsis

```
// In header: <boost/xpressive/xpressive_fwd.hpp>

BOOST_XPRESSIVE_HAS_MS_STACK_GUARD
```

Header <boost/xpressive/xpressive_static.hpp>

Includes everything you need to write static regular expressions and use them.

Header <boost/xpressive/xpressive_typeof.hpp>

Type registrations so that xpressive can be used with the Boost.Typeof library.

Acknowledgments

I am indebted to Joel de Guzman and Hartmut Kaiser for their expert advice during the early states of xpressive's development. Much of static xpressive's syntax is owes a large debt to Spirit, including the syntax for xpressive's semantic actions. I am thankful for John Maddock's excellent work on his proposal to add regular expressions to the standard library, and for various ideas borrowed liberally from his regex implementation. I'd also like to thank Andrei Alexandrescu for his input regarding the behavior of nested regex objects, and Dave Abrahams for his suggestions regarding the regex domain-specific embedded language. Noel Belcourt helped porting xpressive to the Metrowerks CodeWarrior compiler. Markus Schöpflin helped to track down a bug on HP Tru64, and Steven Watanabe suggested the fix.

Special thanks are due to David Jenkins who contributed both ideas, code and documentation for xpressive's semantic actions, symbol tables and attributes. Xpressive's ternary search trie implementation is David's, as is the number parser example in `libs/xpressive/example/numbers.cpp` and the documentation for symbol tables and attributes.

Thanks to John Fletcher for helping track down a runtime assertion when using xpressive with Howard Hinnant's most excellent libc++.

Finally, I would like to thank Thomas Witt for acting as xpressive's review manager.

Appendices

Appendix 1: History

Version 2.1.0 6/12/2008

New Features:

- `skip()` primitive for static regexes, which allows you to specify parts of the input string to ignore during regex matching.
- Range-based `regex_replace()` algorithm interface.
- `regex_replace()` accepts formatter objects and formatter lambda expressions in addition to format strings.

Bugs Fixed:

- Semantic actions in look-aheads, look-behinds and independent sub-expressions execute eagerly instead of causing a crash.

Version 2.0.1 10/23/2007

Bugs Fixed:

- `sub_match<>` constructor copies singular iterator causing debug assert.

Version 2.0.0, 10/12/2007

New Features:

- Semantic actions
- Custom assertions
- Named captures
- Dynamic regex grammars
- Recursive dynamic regexes with `(?R)` construct
- Support for searching non-character data
- Better errors for invalid static regexes
- Range-based regex algorithm interface
- `match_flag_type::format_perl`, `match_flag_type::format_sed`, and `match_flag_type::format_all`
- `operator+(std::string, sub_match<>)` and variants
- Version 2 regex traits get `tolower()` and `toupper()`

Bugs Fixed:

- Complementing single-character sets like `~(set='a')` works.

Version 1.0.2, April 27, 2007

Bugs Fixed:

- Back-references greater than nine work as advertized.

This is the version that shipped as part of Boost 1.34.

Version 1.0.1, October 2, 2006

Bugs Fixed:

- `match_results::position()` works for nested results.

Version 1.0.0, March 16, 2006

Version 1.0!

Version 0.9.6, August 19, 2005

The version reviewed for acceptance into Boost. The review began September 8, 2005. Xpressive was accepted into Boost on September 28, 2005.

Version 0.9.3, June 30, 2005

New Features:

- TR1-style regex_traits interface
- Speed enhancements
- `syntax_option_type::ignore_white_space`

Version 0.9.0, September 2, 2004

New Features:

- It sort of works.

Version 0.0.1, November 16, 2003

Announcement of xpressive: http://lists.boost.org/Archives/boost/2003/11/56312.php

Appendix 2: Not Yet Implemented

The following features are planned for xpressive 2.X:

- `syntax_option_type::collate`
- Collation sequences such as `[.a.]`
- Equivalence classes like `[=a=]`
- Control of nested results generation with `syntax_option_type::nosubs`, and a `nosubs()` modifier for static xpressive.

Here are some wish-list features. You or your company should consider hiring me to implement them!

- Optimized DFA back-end for simple, fast regexing.
- Different regex compiler front ends for basic, extended, awk, grep and egrep regex syntax.

- Fine-grained control over the dynamic regex syntax
- Optional integration with ICU for full Unicode support.
- Improved localization support, possibly as a custom facet for `std::locale`.

Appendix 3: Differences from Boost.Regex

Since many of xpressive's users are likely to be familiar with the Boost.Regex library, I would be remiss if I failed to point out some important differences between xpressive and Boost.Regex. In particular:

- `xpressive::basic_regex<>` is a template on the iterator type, not the character type.
- `xpressive::basic_regex<>` cannot be constructed directly from a string; rather, you must use `basic_regex::compile()` or `regex_compiler<>` to build a regex object from a string.
- `xpressive::basic_regex<>` does not have an `imbue()` member function; rather, the `imbue()` member function is in the `xpressive::regex_compiler<>` factory.
- `boost::basic_regex<>` has a subset of `std::basic_string<>`'s members. `xpressive::basic_regex<>` does not. The members lacking are: `assign()`, `operator[]()`, `max_size()`, `begin()`, `end()`, `size()`, `compare()`, and `operator=(std::basic_string<>)`.
- Other member functions that exist in `boost::basic_regex<>` but do not exist in `xpressive::basic_regex<>` are: `set_expression()`, `get_allocator()`, `imbue()`, `getloc()`, `getflags()`, and `str()`.
- `xpressive::basic_regex<>` does not have a RegexTraits template parameter. Customization of regex syntax and localization behavior will be controlled by `regex_compiler<>` and a custom regex facet for `std::locale`.
- `xpressive::basic_regex<>` and `xpressive::match_results<>` do not have an Allocator template parameter. This is by design.
- `match_not_dot_null` and `match_not_dot_newline` have moved from the `match_flag_type` enum to the `syntax_option_type` enum, and they have changed names to `not_dot_null` and `not_dot_newline`.
- The following `syntax_option_type` enumeration values are not supported: `escape_in_lists`, `char_classes`, `intervals`, `limited_ops`, `newline_alt`, `bk_plus_qm`, `bk_braces`, `bk_parens`, `bk_refs`, `bk_vbar`, `use_except`, `failbit`, `literal`, `perlex`, `basic`, `extended`, `emacs`, `awk`, `grep` ,`egrep`, `sed`, `JavaScript`, `JScript`.
- The following `match_flag_type` enumeration values are not supported: `match_not_bob`, `match_not_eob`, `match_perl`, `match_posix`, and `match_extra`.

Also, in the current implementation, the regex algorithms in xpressive will not detect pathological behavior and abort by throwing an exception. It is up to you to write efficient patterns that do not behave pathologically.

Appendix 4: Performance Comparison

The performance of xpressive is competitive with Boost.Regex. I have run performance benchmarks comparing static xpressive, dynamic xpressive and Boost.Regex on two platforms: gcc (Cygwin) and Visual C++. The tests include short matches and long searches. For both platforms, xpressive comes off well on short matches and roughly on par with Boost.Regex on long searches.

<disclaimer> As with all benchmarks, the true test is how xpressive performs with *your* patterns, *your* input, and *your* platform, so if performance matters in your application, it's best to run your own tests. </disclaimer>

xpressive vs. Boost.Regex with GCC (Cygwin)

Below are the results of a performance comparison between:

- static xpressive

- dynamic xpressive
- Boost.Regex

Test Specifications

Hardware:	hyper-threaded 3GHz Xeon with 1Gb RAM
Operating System:	Windows XP Pro + Cygwin
Compiler:	GNU C++ version 3.4.4 (Cygwin special)
C++ Standard Library:	GNU libstdc++ version 3.4.4
Boost.Regex Version:	1.33+, BOOST_REGEX_USE_CPP_LOCALE, BOOST_REGEX_RECURSIVE
xpressive Version:	0.9.6a

Comparison 1: Short Matches

The following tests evaluate the time taken to match the expression to the input string. For each result, the top number has been normalized relative to the fastest time, so 1.0 is as good as it gets. The bottom number (in parentheses) is the actual time in seconds. The best time has been marked in green.

static xpressive	dynamic xpressive	Boost	Text	Expression
1 (8.79e-07s)	1.08 (9.54e-07s)	2.51 (2.2e-06s)	100- this is a line of ftp response which contains a message string	`^([0-9]+)(\-||$)(.*)$`
1.06 (1.07e-06s)	1 (1.01e-06s)	4.01 (4.06e-06s)	1234-5678-1234-456	`([[:digit:]]{4}[-])){3}[[:digit:]]{3,4}`
1 (1.4e-06s)	1.13 (1.58e-06s)	2.89 (4.05e-06s)	john_maddock@compuserve.com	`^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$`
1 (1.28e-06s)	1.16 (1.49e-06s)	3.07 (3.94e-06s)	foo12@foo.edu	`^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$`
1 (1.22e-06s)	1.2 (1.46e-06s)	3.22 (3.93e-06s)	bob.smith@foo.tv	`^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$`
1.04 (8.64e-07s)	1 (8.34e-07s)	2.5 (2.09e-06s)	EH10 2QQ	`^[a-zA-Z]{1,2}[0-9][0-9A-Za-z]{0,1}{0,1}[0-9][A-Za-z]{2}$`
1.11 (9.09e-07s)	1 (8.19e-07s)	2.47 (2.03e-06s)	G1 1AA	`^[a-zA-Z]{1,2}[0-9][0-9A-Za-z]{0,1}{0,1}[0-9][A-Za-z]{2}$`
1.12 (9.38e-07s)	1 (8.34e-07s)	2.5 (2.08e-06s)	SW1 1ZZ	`^[a-zA-Z]{1,2}[0-9][0-9A-Za-z]{0,1}{0,1}[0-9][A-Za-z]{2}$`

static xpressive	dynamic xpressive	Boost	Text	Expression
1 (7.9e-07s)	1.06 (8.34e-07s)	2.49 (1.96e-06s)	4/1/2001	`^ [ [ : d i - git:]]{1,2}/[[:di-git:]]{1,2}/[[:di-git:]]{4}$`
1 (8.19e-07s)	1.04 (8.49e-07s)	2.4 (1.97e-06s)	12/12/2001	`^ [ [ : d i - git:]]{1,2}/[[:di-git:]]{1,2}/[[:di-git:]]{4}$`
1.09 (8.95e-07s)	1 (8.19e-07s)	2.4 (1.96e-06s)	123	`^ [ - + ] ? [ [ : d i - git:]]*\.?[[:di-git:]]*$`
1.11 (8.79e-07s)	1 (7.9e-07s)	2.57 (2.03e-06s)	+3.14159	`^ [ - + ] ? [ [ : d i - git:]]*\.?[[:di-git:]]*$`
1.09 (8.94e-07s)	1 (8.19e-07s)	2.47 (2.03e-06s)	-3.14159	`^ [ - + ] ? [ [ : d i - git:]]*\.?[[:di-git:]]*$`

Comparison 2: Long Searches

The next test measures the time to find *all* matches in a long English text. The text is the complete works of Mark Twain, from Project Gutenberg. The text is 19Mb long. As above, the top number is the normalized time and the bottom number is the actual time. The best time is in green.

static xpressive	dynamic xpressive	Boost	Expression
1 (0.0263s)	1 (0.0263s)	1.78 (0.0469s)	`Twain`
1 (0.0234s)	1 (0.0234s)	1.79 (0.042s)	`Huck[[:alpha:]]+`
1.84 (1.26s)	2.21 (1.51s)	1 (0.687s)	`[[:alpha:]]+ing`
1.09 (0.192s)	2 (0.351s)	1 (0.176s)	`^[^ ]*?Twain`
1.41 (0.08s)	1.21 (0.0684s)	1 (0.0566s)	`Tom|Sawyer|Huckle-berry|Finn`
1.56 (0.195s)	1.12 (0.141s)	1 (0.125s)	`(Tom|Sawyer|Huckle-berry|Finn).{0,30}river.{0,30}(Tom|Sawyer|Huckleberry|Finn)`

xpressive vs. Boost.Regex with Visual C++

Below are the results of a performance comparison between:

- static xpressive
- dynamic xpressive
- Boost.Regex

Test Specifications

Hardware:	hyper-threaded 3GHz Xeon with 1Gb RAM
Operating System:	Windows XP Pro
Compiler:	Visual C++ .NET 2003 (7.1)
C++ Standard Library:	Dinkumware, version 313
Boost.Regex Version:	1.33+, BOOST_REGEX_USE_CPP_LOCALE, BOOST_REGEX_RECURSIVE
xpressive Version:	0.9.6a

Comparison 1: Short Matches

The following tests evaluate the time taken to match the expression to the input string. For each result, the top number has been normalized relative to the fastest time, so 1.0 is as good as it gets. The bottom number (in parentheses) is the actual time in seconds. The best time has been marked in green.

static xpressive	dynamic xpressive	Boost	Text	Expression
1 (3.2e-007s)	1.37 (4.4e-007s)	2.38 (7.6e-007s)	100- this is a line of ftp response which contains a message string	`^([0-9]+)(\-||$)(.*)$`
1 (6.4e-007s)	1.12 (7.15e-007s)	1.72 (1.1e-006s)	1234-5678-1234-456	`([[:digit:]]{4}[-]){3}[[:digit:]]{3,4}`
1 (9.82e-007s)	1.3 (1.28e-006s)	1.61 (1.58e-006s)	john_maddock@compuserve.com	`^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$`
1 (8.94e-007s)	1.3 (1.16e-006s)	1.7 (1.52e-006s)	foo12@foo.edu	`^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$`
1 (9.09e-007s)	1.28 (1.16e-006s)	1.67 (1.52e-006s)	bob.smith@foo.tv	`^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$`
1 (3.06e-007s)	1.07 (3.28e-007s)	1.95 (5.96e-007s)	EH10 2QQ	`^[a-zA-Z]{1,2}[0-9][0-9A-Za-z]{0,1}{0,1}[0-9][A-Za-z]{2}$`
1 (3.13e-007s)	1.09 (3.42e-007s)	1.86 (5.81e-007s)	G1 1AA	`^[a-zA-Z]{1,2}[0-9][0-9A-Za-z]{0,1}{0,1}[0-9][A-Za-z]{2}$`
1 (3.2e-007s)	1.09 (3.5e-007s)	1.86 (5.96e-007s)	SW1 1ZZ	`^[a-zA-Z]{1,2}[0-9][0-9A-Za-z]{0,1}{0,1}[0-9][A-Za-z]{2}$`

static xpressive	dynamic xpressive	Boost	Text	Expression
1 (2.68e-007s)	1.22 (3.28e-007s)	2 (5.36e-007s)	4/1/2001	`^ [ [ : d i - git:]]{1,2}/[[:di-git:]]{1,2}/[[:di-git:]]{4}$`
1 (2.76e-007s)	1.16 (3.2e-007s)	1.94 (5.36e-007s)	12/12/2001	`^ [ [ : d i - git:]]{1,2}/[[:di-git:]]{1,2}/[[:di-git:]]{4}$`
1 (2.98e-007s)	1.03 (3.06e-007s)	1.85 (5.51e-007s)	123	`^[-+]?[[:di-git:]]*\.?[[:di-git:]]*$`
1 (3.2e-007s)	1.12 (3.58e-007s)	1.81 (5.81e-007s)	+3.14159	`^[-+]?[[:di-git:]]*\.?[[:di-git:]]*$`
1 (3.28e-007s)	1.11 (3.65e-007s)	1.77 (5.81e-007s)	-3.14159	`^[-+]?[[:di-git:]]*\.?[[:di-git:]]*$`

Comparison 2: Long Searches

The next test measures the time to find *all* matches in a long English text. The text is the complete works of Mark Twain, from Project Gutenberg. The text is 19Mb long. As above, the top number is the normalized time and the bottom number is the actual time. The best time is in green.

static xpressive	dynamic xpressive	Boost	Expression
1 (0.019s)	1 (0.019s)	2.98 (0.0566s)	`Twain`
1 (0.0176s)	1 (0.0176s)	3.17 (0.0556s)	`Huck[[:alpha:]]+`
3.62 (1.78s)	3.97 (1.95s)	1 (0.492s)	`[[:alpha:]]+ing`
2.32 (0.344s)	3.06 (0.453s)	1 (0.148s)	`^[^ ]*?Twain`
1 (0.0576s)	1.05 (0.0606s)	1.15 (0.0664s)	`Tom|Sawyer|Huckle-berry|Finn`
1.24 (0.164s)	1.44 (0.191s)	1 (0.133s)	`(Tom|Sawyer|Huckle-berry|Finn).{0,30}river.{0,30}(Tom|Sawyer|Huckleberry|Finn)`

Appendix 5: Implementation Notes

Cycle collection with `tracking_ptr<>`

In xpressive, regex objects can refer to each other and themselves by value or by reference. In addition, they ref-count their referenced regexes to keep them alive. This creates the possibility for cyclic reference counts, and raises the possibility of memory leaks. xpressive avoids leaks by using a type called `tracking_ptr<>`. This doc describes at a high level how `tracking_ptr<>` works.

Constraints

Our solution must meet the following design constraints:

- No dangling references: All objects referred to directly or indirectly must be kept alive as long as the references are needed.
- No leaks: all objects must be freed eventually.
- No user intervention: The solution must not require users to explicitly invoke some cycle collection routine.
- Clean-up is no-throw: The collection phase will likely be called from a destructor, so it must never throw an exception under any circumstance.

Handle-Body Idiom

To use `tracking_ptr<>`, you must separate your type into a handle and a body. In the case of xpressive, the handle type is called `basic_regex<>` and the body is called `regex_impl<>`. The handle will store a `tracking_ptr<>` to the body.

The body type must inherit from `enable_reference_tracking<>`. This gives the body the bookkeeping data structures that `tracking_ptr<>` will use. In particular, it gives the body:

1. `std::set<shared_ptr<body> > refs_` : collection of bodies to which this body refers, and
2. `std::set<weak_ptr<body> > deps_` : collection of bodies which refer to this body.

References and Dependencies

We refer to (1) above as the "references" and (2) as the "dependencies". It is crucial to the understanding of `tracking_ptr<>` to recognize that the set of references includes both those objects that are referred to directly as well as those that are referred to indirectly (that is, through another reference). The same is true for the set of dependencies. In other words, each body holds a ref-count directly to every other body that it needs.

Why is this important? Because it means that when a body no longer has a handle referring to it, all its references can be released immediately without fear of creating dangling references.

References and dependencies cross-pollinate. Here's how it works:

1. When one object acquires another as a reference, the second object acquires the first as a dependency.
2. In addition, the first object acquires all of the second object's references, and the second object acquires all of the first object's dependencies.
3. When an object picks up a new reference, the reference is also added to all dependent objects.
4. When an object picks up a new dependency, the dependency is also added to all referenced objects.
5. An object is never allowed to have itself as a dependency. Objects may have themselves as references, and often do.

Consider the following code:

```
sregex expr;
{
    sregex group  = '(' >> by_ref(expr) >> ')';                 // (1)
    sregex fact   = +_d | group;                                // (2)
    sregex term   = fact >> *(('*' >> fact) | ('/' >> fact));   // (3)
    expr          = term >> *(('+' >> term) | ('-' >> term));   // (4)
}                                                               // (5)
```

Here is how the references and dependencies propagate, line by line:

Expression	Effects
1) `sregex group = '(' >> by_ref(expr) >> ')';`	`group: cnt=1 refs={expr} deps={}` `expr: cnt=2 refs={} deps={group}`

2) `sregex fact = +_d | group;`	`group: cnt=2 refs={expr} deps={fact}`
	`expr: cnt=3 refs={} deps={group,fact}`
	`fact: cnt=1 refs={expr,group} deps={}`

3) `sregex term = fact >> *(('*' >> fact) | ('/' >> fact));`	`group: cnt=3 refs={expr} deps={fact,term}`
	`expr: cnt=4 refs={} deps={group,fact,term}`
	`fact: cnt=2 refs={expr,group} deps={term}`
	`term: cnt=1 refs={expr,group,fact} deps={}`

4) `expr = term >> *(('+' >> term) | ('-' >> term));`	`group: cnt=5 refs={expr,group,fact,term} deps={expr,fact,term}`
	`expr: cnt=5 refs={expr,group,fact,term} deps={group,fact,term}`
	`fact: cnt=5 refs={expr,group,fact,term} deps={expr,group,term}`
	`term: cnt=5 refs={expr,group,fact,term} deps={expr,group,fact}`

5) `}`	`expr: cnt=2 refs={expr,group,fact,term} deps={group,fact,term}`

This shows how references and dependencies propagate when creating cycles of objects. After line (4), which closes the cycle, every object has a ref-count on every other object, even to itself. So how does this not leak? Read on.

Cycle Breaking

Now that the bodies have their sets of references and dependencies, the hard part is done. All that remains is to decide when and where to break the cycle. That is the job of `tracking_ptr<>`, which is part of the handle. The `tracking_ptr<>` holds 2 `shared_ptr`s. The first, obviously, is the `shared_ptr<body>` -- the reference to the body to which this handle refers. The other `shared_ptr` is used to break the cycle. It ensures that when all the handles to a body go out of scope, the body's set of references is cleared.

This suggests that more than one handle can refer to a body. In fact, `tracking_ptr<>` gives you copy-on-write semantics -- when you copy a handle, the body is shared. That makes copies very efficient. Eventually, all the handles to a particular body go out of scope. When that happens, the ref count to the body might still be greater than 0 because some other body (or this body itself!) might be holding a reference to it. However, we are certain that the cycle-breaker's ref-count goes to 0 because the cycle-breaker only lives in handles. No more handles, no more cycle-breakers.

What does the cycle-breaker do? Recall that the body has a set of references of type `std::set<shared_ptr<body> >`. Let's call this type "references_type". The cycle-breaker is a `shared_ptr<references_type>`. It uses a custom deleter, which is defined as follows:

```
template<typename DerivedT>
struct reference_deleter
{
    void operator ()(std::set<shared_ptr<DerivedT> > *refs) const
    {
        refs->clear();
    }
};
```

The job of to the cycle breaker is to ensure that when the last handle to a body goes away, the body's set of references is cleared. That's it.

We can clearly see how this guarantees that all bodies are cleaned up eventually. Once every handle has gone out of scope, all the bodies' sets of references will be cleared, leaving none with a non-zero ref-count. No leaks, guaranteed.

It's a bit harder to see how this guarantees no dangling references. Imagine that there are 3 bodies: A, B and C. A refers to B which refers to C. Now all the handles to B go out of scope, so B's set of references is cleared. Doesn't this mean that C gets deleted, even though it is being used (indirectly) by A? It doesn't. This situation can never occur because we propagated the references and dependencies above such that A will be holding a reference directly to C in addition to B. When B's set of references is cleared, no bodies get deleted, because they are all still in use by A.

Future Work

All these `std::sets` and `shared_ptrs` and `weak_ptrs`! Very inefficient. I used them because they were handy. I could probably do better.

Also, some objects stick around longer than they need to. Consider:

```
sregex b;
{
    sregex a = _;
    b = by_ref(a);
    b = _;
}
// a is still alive here!
```

Due to the way references and dependencies are propagated, the `std::set` of references can only grow. It never shrinks, even when some references are no longer needed. For xpressive this isn't an issue. The graphs of referential objects generally stay small and isolated. If someone were to try to use `tracking_ptr<>` as a general ref-count-cycle-collection mechanism, this problem would have to be addressed.