Boost.Xpressive

Eric Niebler

翻訳にあたって

- 本書は Boost.Xpressive ドキュメントの日本語訳です。原文書のバージョンは翻訳時の最新である 1.55.0 です。
- 原文の誤りは修正したうえで翻訳しました。
- 外部文書の表題等は英語のままにしてあります。
- 原文に含まれているローカルファイルへのハイパーリンクは削除しています。
- 文中の正規表現、部分式、書式化文字列は `regular-expression` のように記します。
- マッチ対象の入力テキストは `"input-text"` のように記します。
- ファイル名、ディレクトリ名は `pathname` のように記します。
- その他、読みやすくするためにいくつか書式の変更があります。
- 翻訳の誤り等は exeal に連絡ください。

前口上[2]

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

説明

xpressive は正規表現の高度な C++オブジェクト指向テンプレートライブラリである。正規表現（正規式）を文字列で渡して実行時に解析することもできれば、式テンプレートを使ってコンパイル時に解析することもできる。正規表現はお互いを参照しあうことも、自身を再帰的に参照することもでき、これらから複雑な文法を任意に構築できる。

---

1 訳注 原著のライセンス: *"Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)"*。本稿のライセンスも同様とします。

2 訳注 http://snltranscripts.jt.org/75/75ishimmer.phtml とか。検索すると動画も出てきます。

動機

C++でテキスト処理をする場合、2 つの別々な選択肢がある。正規表現エンジンとパーサジェネレータである。正規表現エンジン（Boost.Regex 等）は強力で柔軟性がある。パターンは文字列で表現し、実行時に指定する。しかしながら構文エラーもまた、実行時まで検出されない。また正規表現は、入れ子かつ数の合った HTML タグに対するマッチのような、より高度なテキスト処理問題には適していない。従来、こういった処理はパーサジェネレータ（Spirit パーサジェネレータ等）により取り扱われてきた。こうした怪物級の連中はより強力だが柔軟性がなく、通常はその場での文法規則の任意修正を許していない。さらに言えば、正規表現の網羅的なバックトラックのセマンティクス（意味）も有していないため、パターンの種類によっては大変な労力が必要になる場合がある。

xpressive はこれら 2 つのアプローチを相互にシームレスに接続し、C++テキスト処理の世界において固有の地位を築く。xpressive を使えば、Boost.Regex を使うのと同様に正規表現を文字列で表現できる。あるいは Spirit を使うのと同様に正規表現を C++の式として記述でき、テキスト処理専用の組み込み言語による恩恵を受けることができる。さらに、この 2 つを混ぜて両方の利点を引き出すこともできる。つまり正規表現**文法**を、あるときは静的に束縛（ハードコードし、コンパイラによる構文チェックが可能）、またあるときは動的に束縛し実行時に記述できる。これら 2 種類の正規表現はお互いに再帰的に参照可能で、普通の正規表現では不可能なマッチが可能である。

影響を受けたか関係のあるライブラリ

xpressive のインターフェイスの設計については、John Maddock の Boost.Regex と、正規表現が標準ライブラリに追加されることになった彼の草案に多大な影響を受けた。また、静的 xpressive のモデルとなった Joel de Guzman の Spirit パーサフレームワークにも大きなインスピレーションを受けた。他にインスピレーションを受けたものとして、Perl 6 の設計と GRETA がある（正規表現の慣習を変える Perl 6 の変更について、概要がここにある）。

ユーザーガイド

本節では xpressive を使ったテキスト処理、パース処理の方法を説明する。xpressive の特定のコンポーネントについて詳細な情報を探している場合は、リファレンスの節を見よ。

はじめに

xpressive とは何か

xpressive は正規表現のテンプレートライブラリである。正規表現・正規式（regex とも[3]）は実行時に動的に解析される文字列としても（動的正規表現）、またはコンパイル時に解析される式テンプレート[4]としても（静的正規表現）記述できる。動的正規表現の利点は、実行時にユーザーが入力したり、初期化ファイルから読み取りが可能なことである。静的正規表現には利点がいくつかある。文字列ではなく C++式テンプレートなのでコンパイル時に構文チェックを受ける。また、プログラム内のコードとデータを参照可能なので、正規表現マッチの最中にコードを呼び出すこともできる。加えて静的束縛されるので、コンパイラは静的正規表現についてより高速なコードを生成する可能性がある。

xpressive のこの 2 本立ての機能は独特かつ強力である。静的 xpressive は Spirit パーサフレームワークのようなものである。Spirit と同様、式テンプレートを使った静的正規表現で文法を構築できる（Spirit と異なり、xpressive はパターンマッチを探索するためにあらゆる可能性を試行する網羅的なバックトラックを行う）。動的正規表現は Boost.Regex のようなものである。実際、xpressive のインターフェイスは Boost.Regex を使ったことのある人にとっては親しみやすいはずである。xpressive の革新的な点は、静的正規表現と動的正規表現を同じプログラム内（同じ式内でも！）で混ぜてマッチできることである。動的正規表現を静的正規表現に組み込むこともその逆も可能である。組み込んだ正規表現はパターンマッチに必要な検索やバックトラックに対して完全に機能する。

Hello, world!

理論は十分だ。xpressive スタイルの *Hello World* を見よう。

```
#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'; // マッチ全体
        std::cout << what[1] << '\n'; // 1 番目の捕捉
```

3 訳注 Regular expression の省略形ですが、翻訳版では省略せず「正規表現」「正規式」と書きます。

4 Expression Templates（英語）を参照。

```
        std::cout << what[2] << '\n'; // 2番目の捕捉
    }

    return 0;
}
```

このプログラムは以下を出力する。

```
hello world!
hello
world
```

このコードでまず注意すべきは、xpressive の型がすべて boost::xpressive 名前空間にあるということである。

注意

本文書における残りのほとんどの例では using namespace boost::xpressive; ディレクティブを省略しているが、実際には必要である。

次に注意すべきは正規表現オブジェクトの型が sregex ということである。Boost.Regex に馴染んでいるのであれば、今まで使っていたものとは違うという点に気をつけなければならない。“sregex” の “s” は “string” のことであり、この正規表現は std::string オブジェクト内でパターンを探索するのに使用するということを表している。この違いとそれが意味するところについては後で述べる。

正規表現オブジェクトをどのように初期化するかに注目する。

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

正規表現オブジェクトを文字列から作成する場合、basic_regex<>::compile()といったファクトリメソッドを呼び出さなければならない。これもまた、xpressive が他のオブジェクト指向正規表現ライブラリと異なっている点である。他のライブラリでは正規表現は文字列の強化版のような扱いだが、xpressive では正規表現は文字列ではなく、ドメイン固有言語における小さなプログラムである。文字列はそのような言語の**表現**の 1 つにすぎない。もう 1 つの表現が式テンプレートである。例えば上のコード行は以下と等価である。

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

これは同じ正規表現を表しているが、静的 xpressive が定義するドメイン固有の組み込み言語を用いている点が異なる。

見てのとおり、静的正規表現の構文には標準的な Perl の構文と顕著に違う点がある。これは C++構文の制約によるもので、最も大きな違いは「後続」を表す>>の使用である。例えば Perl では部分式を続けて書くことができる。

```
abc
```

しかし C++では部分式を分離する演算子がなければならない。

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

Perl では括弧`()`は特別な意味をもつ。これらはグループ化を行うが、`$1`や`$2`といった後方参照を作成するという副作用がある。C++では括弧を多重定義して副作用を与えることはできない。そこで同じ効果を得るために`s1`や`s2`という特殊なトークンを使用する。これらに代入を行うことで後方参照を作成する（xpressive では部分マッチ（sub-match）という）。

他に注意すべき点として、1 回以上の繰り返しを表す+演算子の位置が後置から前置になっているということがある。これは C++が後置の+演算子をもたないためである。よって、

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

これは以下と同じである。

```
+_w
```

他のすべての違いについては後で触れる。

xpressive のインストール

xpressive の入手

xpressive の入手方法は 2 つある。第 1 のより簡単な方法は Boost の最新版をダウンロードすることである。http://sf.net/projects/boost へ行き、“Download” リンクをたどるだけである。

2 番目の方法は Boost の Subversion リポジトリに直接アクセスすることである。http://svn.boost.org/trac/boost へ行き、そこにある匿名 Subversion アクセス方法に従うとよい。Boost Subversion にあるのは不安定版である。

xpressive を使ったビルド

xpressive はヘッダのみのテンプレートライブラリであり、あなたのビルドスクリプトを書き直したり個別のライブラリファイルにリンクする必要はない。`#include <boost/xpressive/xpressive.hpp>` とするだけでよい。使用するのが静的正規表現だけであれば、`xpressive_static.hpp`だけをインクルードすることでコンパイル時間を短縮できる。同様に動的正規表現だけを使用するのであれば`xpressive_dynamic.hpp`をインクルードするとよい。

静的正規表現とともに意味アクションやカスタム表明を使用したければ、`regex_actions.hpp`も追加でインクルードする必要がある。

必要要件

xpressive を使用するには Boost 1.34.1 以降が必要である。

サポートするコンパイラ

現在のところ、Boost.Xpressive は以下のコンパイラで動作する。

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

Boost の退行テスト結果のページにある最新テスト結果を参照するとよい。

質問、コメント、バグ報告は eric <at> boost-consulting <dot> com に送ってほしい。

クイックスタート

xpressive で何かするのに知っておくべきことはそう多くない。xpressive が提供する型とアルゴリズムの 5 セント旅行に出かけよう。

表 1: xpressive のツールボックス

ツール	説明
basic_regex<>	コンパイル済みの正規表現を保持する。basic_regex<>は xpressive で最も重要な型である。xpressive で何かする場合は basic_regex<>型のオブジェクトを作成することから始める。
match_results<>、sub_match<>	match_results<> は、regex_match() や regex_search() 操作の結果を保持する。sub_match<>オブジェクトのベクタのように振舞う。個別の sub_match<>オブジェクトはマーク済み部分式（Perl における後方参照）を保持する。基本的にはマーク済み部分式の開始と終了を表すイテレータの組にすぎない。
regex_match()	文字列が正規表現にマッチするか調べる。regex_match()が成功するのは、文字列全体の先頭から終端までが正規表現にマッチする場合である。regex_match()に match_results<>を与えると、見つかったマーク済み部分式が書き込まれる。
regex_search()	正規表現にマッチする部分文字列を文字列内で検索する。regex_search()は文字列内のあらゆる位置でマッチを検索する。文字列の先頭から開始し、マッチを見つけるか文字列内をすべて走査すると終了する。regex_match()と同様、regex_search()に match_results<>を与えると、見つかったマーク済み部分式が書き込まれる。
regex_replace()	入力文字列、正規表現、置換文字列を与えると、regex_replace()は入力文字列内の正規表現にマッチした部分を置換文字列で置換した新しい文字列を構築する。置換文字列にはマーク済み部分式への参照を含めることができる。
regex_iterator<>	文字列内の正規表現にマッチする位置を見つける STL 互換のイテレータ。regex_iterator<>を参照はがしすると match_results<>が返る。regex_iterator<>をインクリメントすると次のマッチを検索する。
regex_token_iterator<>	regex_iterator<>と似ているが、regex_token_iterator<>を参照はがしすると文字列が

	返る。既定では正規表現にマッチした部分文字列全体が返るが、一度にいずれかあるいはすべてのマーク済み部分式を 1 つずつ返すように設定することもできる。また、文字列の正規表現にマッチ**しなかった**部分を返すよう設定することもできる。
regex_compiler<>	basic_regex<>オブジェクトのファクトリ。文字列を正規表現に「コンパイル」する。basic_regex<>クラスは内部で regex_compiler<>を使用するファクトリメソッドをもっているので、大抵の場合 regex_compiler<>を直接取り扱う必要はない。しかし、basic_regex<>オブジェクトを異なる std::locale で作成するなど変わったことをする必要がある場合は regex_compiler<>を明示的に使用しなければならない。

xpressive が提供するツール群について少しは分かったと思う。次の 2 つの質問に答えれば正しいツールを選択できるだろう。

1. データを走査するのに使う**イテレータ**の型は何か。
2. データを使って何を**したい**のか。

イテレータの型

xpressive において、ほとんどのクラスはイテレータ型を引数にもつテンプレートである。正しい型を簡単に選択できるように xpressive は共通の typedef をいくつか定義している。以下の表を見ればイテレータ型から正しい型が分かる。

表 2: xpressive の typedef とイテレータ型の対応

	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

機械的な名前付け規約に注意していただきたい。これらの型の多くは一緒に使用するため、名前付け規約は一貫性という点で助けになる。例えば sregex があれば一緒に使うのは smatch という具合である。

これら 4 つのイテレータ型以外については、テンプレートを直接使用しイテレータ型を指定するとよい。

タスク

パターンを使うのは 1 度か、複数回か。検索か置換か。xpressive はこれらをすべてカバーし、他にも多くの機能がある。以下が早見表である。

表 3: 処理とツール

次を行うには…	**以下を使用せよ**
文字列全体が正規表現にマッチするか調べる	regex_match()アルゴリズム
文字列が正規表現にマッチする部分文字列を含むか調べる	regex_search()アルゴリズム

正規表現にマッチする部分文字列をすべて置換する	regex_replace()アルゴリズム
正規表現にマッチする部分文字列をすべて検索し、1 つずつたどる	regex_iterator<>クラス
それぞれが正規表現にマッチするトークンに文字列を分割する	regex_token_iterator<>クラス
正規表現を区切りとして文字列を分割する	regex_token_iterator<>クラス

これらのアルゴリズムとクラスの厄介な詳細はリファレンスの節で述べる。

ヒント

上の表の各処理をクリックすると、xpressive を使った完全なプログラム例が表示される。

正規表現オブジェクトの作成

xpressive を使う場合、最初に行うのが basic_regex<>オブジェクトの作成である。本節では静的・動的の 2 つの表現方法による正規表現作成の基本を見ていく。

静的正規表現

概要

xpressive が他の C/C++正規表現ライブラリと一線を画すのは、C++の式を用いて正規表現を記述する機能による。xpressive は演算子の多重定義と**式テンプレート**という技術を使って、パターンマッチのための小言語を C++に組み込むことでこれを実現している。これら「静的正規表現」には文字列ベースのものに比較して多くの利点がある。特に以下の点を挙げておく。

- コンパイル時に構文がチェックされる。実行時に構文エラーで失敗することがない。
- 他の C++データ、コード、他の正規表現を自然に参照できる。正規表現の外部での文法構築、および正規表現マッチの一部として実行されるユーザー定義アクションの束縛が簡単になる。
- 静的束縛され、インライン化と最適化が促進される。静的正規表現は状態表、仮想関数、バイトコード、関数ポインタによる呼び出しといったコンパイル時に解決できないものを必要としない。
- 検索対象が文字列に限定されない。例えば、数値配列からパターンを探索する静的正規表現を宣言できる。

静的正規表現の組み立ては C++の式を使うので、合法な C++の式規則の制約を受ける。残念ながら、「伝統的な」正規表現構文をすべてきれいに C++に対応させられるわけではない。そういうわけで、無理な対応は試みず C++として合法な構文を用意する。

構築と代入

静的正規表現の作成は、basic_regex<>型のオブジェクトへの代入により行う。例えば、以下は std::string 型のオブジェクトに対してパターンを探索する正規表現を定義する。

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

代入の動作も似たようなものである。

文字と文字列のリテラル

静的正規表現において、文字と文字列リテラルはそれ自身にマッチする。例えば上の正規表現において'$'と'.'は、それぞれ文字'$' 、'.'にマッチする。Perl において $ と . がメタ文字であるからといって混乱しないでいただきたい。xpressive ではリテラルは常にそれ自身を表す。

静的正規表現でリテラルを使用する場合は、少なくとも片方のオペランドはリテラル以外であることに注意しなければならない。例えば以下は正しい正規表現では**ない**。

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

二項>>演算子の 2 つのオペランドが両方ともリテラル、また単項+演算子のオペランドもリテラルになっている。よってこれらの文は組み込みの C++二項右シフト、単項プラス演算子をそれぞれ呼び出す。これは期待した動作ではない。演算子の多重定義が機能するには、少なくとも 1 つのオペランドがユーザー定義型でなければならない。xpressive の as_xpr() ヘルパ関数を使うと式を正規表現の世界に「引き込み」、演算子の多重定義に正しい演算子を見つけるよう強制できる。上の 2 つは次のように書くべきだ。

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

結合と選択

すでに見てきたように、静的正規表現における部分式は結合演算子>>で分離されていなければならない。この演算子は「～の後に」などと読み替えるとよい。[5]

```
// 後ろに数字が続く'a'にマッチ
sregex re = 'a' >> _d;
```

選択（分岐）は|演算子を使用する。Perl と同様の動作をする。この演算子は「または」などと読み替えるとよい。例えば、

```
// 1 文字以上の数字、または単語構成文字にマッチ
sregex re = +( _d | _w );
```

グループ化と捕捉

Perl では括弧 () は特別な意味をもつ。これらはグループ化を行うが、$1 や$2 といった後方参照を作成するという副作用がある。C++では括弧を多重定義して副作用を与えることはできない。そこで同じ効果を得るために s1 や s2 という特殊なトークンを使用する。これらに代入を行うことで後方参照を作成する。後方参照は Perl の\1 や\2 のような使い方で式中で使用できる。例えば以下の

5　訳注　原文は “followed by”。無理に日本語にしないほうがいいかもしれません…。

HTML タグのマッチを探索する正規表現を考えよう。

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

静的正規表現では、次のようになる。

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

s1 への代入により後方参照を捕捉し、パターンの後ろのほうでマッチする終了タグを探索するのに s1 を使っていることに注意していただきたい。

ヒント：後方参照を捕捉せずにグループ化けを行う

xpressive では、後方参照を捕捉せずにグループ化を行うには s1 なしで () を使うだけでよい。これは Perl の捕捉なしのグループ化構造(?:)と等価である。

大文字小文字の区別と国際化

Perl ではパターン修飾子(?i:)を使用することで正規表現が大文字小文字を区別しなくなる。xpressive でも icase という大文字小文字を区別しないパターン修飾子がある。以下のように使用する。

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

この正規表現では、this は大文字小文字を区別するが that は大文字小文字を無視してマッチを行う。

大文字小文字を区別しない正規表現で問題になるのが国際化である。大文字小文字を区別しない文字の比較をどのように行うべきだろうか？ また、多くの文字クラスはロカール（locale）依存である。どの文字が digit にマッチし、どの文字が alpha にマッチするだろうか？ その答えは正規表現オブジェクトが使用する std::locale オブジェクトに依存する。既定ではすべての正規表現オブジェクトはグローバルなロカールを使用する。既定をオーバーライドするには、以下のように imbue() パターン修飾子を使用する。

```
std::locale my_locale = /* std::locale オブジェクトを初期化する */;
sregex re = imbue( my_locale )( +alpha >> +digit );
```

この正規表現は alpha と digit を my_locale にしたがって処理する。正規表現の振る舞いをカスタマイズする方法については地域化と正規表現の特性の節を見よ。

静的 xpressive 構文のカンニングペーパー

一般的な正規表現の構造と静的 xpressive の対応を以下の表に示す。

表 4: Perl の構文と静的 xpressive の構文

Perl	静的 xpressive	意味
`.`	`_`	任意の 1 文字（Perl の `/s` 修飾子が使われているとして）。
`ab`	`a >> b`	`a` および `b` 部分式の結合。
`a|b`	`a | b`	`a` および `b` 部分式の選択。
`(a)`	`(s1= a)`	後方参照のグループ化と捕捉。
`(?:a)`	`(a)`	後方参照の捕捉を伴わないグループ化。
`\1`	`s1`	以前捕捉した後方参照。
`a*`	`*a`	0 回以上の貪欲な繰り返し。
`a+`	`+a`	1 回以上の貪欲な繰り返し。
`a?`	`!a`	0 回か 1 回の貪欲な繰り返し。
`a{n,m}`	`repeat<n,m>(a)`	`n` 回以上 `m` 回以下の貪欲な繰り返し。
`a*?`	`-*a`	0 回以上の貪欲でない繰り返し。
`a+?`	`-+a`	1 回以上の貪欲でない繰り返し。
`a??`	`-!a`	0 回か 1 回の貪欲でない繰り返し。
`a{n,m}?`	`-repeat<n,m>(a)`	`n` 回以上 `m` 回以下の貪欲でない繰り返し。
`^`	`bos`	シーケンスの先頭を表す表明。
`$`	`eos`	シーケンスの終端を表す表明。
`\b`	`_b`	単語境界の表明。
`\B`	`~_b`	単語境界以外の表明。
`\n`	`_n`	リテラルの改行。
`.`	`~_n`	リテラルの改行以外の任意の 1 文字（Perl の `/s` 修飾子が使われていないとして）。
`\r?\n|\r`	`_ln`	論理改行。
`[^\r\n]`	`~_ln`	論理改行以外の任意の 1 文字。
`\w`	`_w`	単語構成文字。`set[alnum | '_']`と同じ。
`\W`	`~_w`	単語構成文字以外。`~set[alnum | '_']`と同じ。
`\d`	`_d`	数字。
`\D`	`~_d`	数字以外。
`\s`	`_s`	空白類文字。
`\S`	`~_s`	空白類文字以外。
`[:alnum:]`	`alnum`	アルファベットおよび数値文字。
`[:alpha:]`	`alpha`	アルファベット文字。
`[:blank:]`	`blank`	水平空白文字。
`[:cntrl:]`	`cntrl`	制御文字。
`[:digit:]`	`digit`	数字。

`[:graph:]`	`graph`	グラフィカルな文字。
`[:lower:]`	`lower`	小文字。
`[:print:]`	`print`	印字可能な文字。
`[:punct:]`	`punct`	区切り文字。
`[:space:]`	`space`	空白類文字。
`[:upper:]`	`upper`	大文字。
`[:xdigit:]`	`xdigit`	16 進数字。
`[0-9]`	`range('0','9')`	`'0'`から`'9'`の範囲の文字。
`[abc]`	`as_xpr('a') | 'b' | 'c'`	`'a'`、`'b'`、または`'c'`のいずれかの文字。
`[abc]`	`(set= 'a','b','c')`	同上。
`[0-9abc]`	`set[ range('0','9') | 'a' | 'b' | 'c' ]`	`'a'`、`'b'`、または`'c'`のいずれか、または`'0'`から`'9'`の範囲の文字。
`[0-9abc]`	`set[ range('0','9') | (set= 'a','b','c') ]`	同上。
`[^abc]`	`~(set= 'a','b','c')`	`'a'`、`'b'`、または`'c'`のいずれでもない文字。
`(?i:`*stuff*`)`	`icase(`*stuff*`)`	*stuff* の大文字小文字を区別しないマッチを行う。
`(?>`*stuff*`)`	`keep(`*stuff*`)`	独立部分式。*stuff* のマッチを行いバックトラックを切る。
`(?=`*stuff*`)`	`before(`*stuff*`)`	肯定先読み表明。*stuff* の前にマッチするが *stuff* 自身はマッチに含まない。
`(?!`*stuff*`)`	`~before(`*stuff*`)`	否定先読み表明。*stuff* の前以外にマッチ。
`(?<=`*stuff*`)`	`after(`*stuff*`)`	肯定後読み表明。*stuff* の後にマッチするが *stuff* 自身はマッチに含まない（*stuff* は固定長でなければならない）。
`(?<!`*stuff*`)`	`~after(`*stuff*`)`	否定後読み表明。

`(?P<name>`*stuff*`)`	`mark_tag` *name*`(`*n*`);`
	…
	`(`*name*`= `*stuff*`)`
		名前付き捕捉を作成。

`(?P=`*name*`)`	`mark_tag` *name*`(`*n*`);` … *name*	作成した名前付き捕捉への後方参照。

動的正規表現

概要

静的正規表現は一級品だが、ときにはもっと別の…、つまり動的正規表現が必要な場合もある。正規表現検索・置換機能を備えたテキストエディタを開発中だとしよう。正規表現は、実行時にエンドユーザーからの入力として受け付けなければならない。文字列を正規表現に解析する方法が必要であり、xpressive の動的正規表現がそれに相当する。これらは静的正規表現と同じコアコンポーネントから構築するが、遅延束縛のため実行時にパターンを指定できる。

構築と代入

動的正規表現を作成する方法は 2 つある。basic_regex<>::compile()関数によるものとregex_compiler<>クラステンプレートによるものである。既定のロカールでよければ basic_regex<>::compile()を使うとよい。別のロカールを指定する必要がある場合は、regex_compiler<>を使用する。正規表現文法の節で、regex_compiler<>の他の使用について述べる。

以下は basic_regex<>::compile()の使用例である。

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

以下は regex_compiler<>を使った同じ例である。

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

basic_regex<>::compile()は regex_compiler<>を使って実装している。

動的 xpressive の構文

動的構文は合法な C++の式規則による制約を受けないので、動的正規表現については慣れ親しんだ構文が使える。そういうわけで動的正規表現については xpressive は、正規表現を標準ライブラリに追加することになった John Maddock の草案に従った。本質的には ECMAScript により標準化された構文であり、国際化のための細かい変更を加えてある。

構文の網羅的な文書は他にあるので、ここでは仕様の複製はせず、既存の標準を参照するにとどめる。

国際化

静的正規表現と同様、動的正規表現の国際化サポートは別の std::locale を指定することによる。これを行うには regex_compiler<>を使用しなければならない。regex_compiler<>クラスは imbue()関数をもつ。regex_compiler<>オブジェクトに対してカスタムの std::locale を使って imbue()を呼び出すと、それ以降に regex_compiler<>でコンパイルした正規表現オブジェクトはそのロカールを使用するようになる。例えば、

```
std::locale my_locale = /* ここでロカールオブジェクトを初期化する */;
sregex_compiler compiler;
compiler.imbue( my_locale );
sregex re = compiler.compile( "\\w+|\\d+" );
```

この正規表現は、組み込みの文字集合\wおよび\dを処理するのに my_locale を使用する。

マッチと検索

概要

正規表現オブジェクトの作成が終わったら、regex_match()および regex_search()アルゴリズムで文字列からパターンを検索

する。本節では正規表現のマッチと検索の基本について述べる。Boost.Regex ライブラリの regex_match() および regex_search()の振る舞いについて理解しているなら、xpressive 版でも同様の動作をすると考えてよい。

文字列が正規表現にマッチするか調べる

regex_match()アルゴリズムは正規表現が与えられた入力にマッチするか調べる。

警告

regex_match()アルゴリズムは、正規表現が**入力全体**の先頭から終端までマッチした場合のみ成功する。正規表現が入力の一部分だけにマッチする場合は regex_match()は偽を返す。文字列から正規表現にマッチする部分文字列を探す場合は、regex_search()アルゴリズムを使うとよい。

入力は std::string、C 形式の null 終端文字列、イテレータの組といった双方向範囲である。いずれの場合でも、入力シーケンスを走査するイテレータ型は正規表現オブジェクトの宣言に使用したイテレータ型と一致していなければならない（イテレータに対する正しい正規表現の型は、クイックスタートの表を見れば分かる）。

```
cregex cre = +_w;  // C形式の文字列にマッチ
sregex sre = +_w;  // std::stringsにマッチ

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

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

if( regex_match( "hello", sre ) )              // エラー！ イテレータが一致していない！
    { /*...*/ }
```

regex_match()アルゴリズムは省略可能な出力引数として match_results<>構造体を受け付ける。この引数が与えられると、regex_match()アルゴリズムは正規表現のどの部分が入力のどの部分にマッチしたかの情報を match_results<>構造体に書き込む。

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

// regex_matchの結果を"what"に格納する
if( regex_match( "hello", what, cre ) )
{
    std::cout << what[1] << '\n'; // "o"を印字する
}
```

regex_match()アルゴリズムはさらに省略可能な引数として match_flag_type ビットマスクを受け付ける。match_flag_type を与えると、マッチをどのように行うかある程度制御できる。このフラグの完全なリストと意味については match_flag_type のリファレンスを見よ。

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

// match_not_bol の意味は、「"bol"（行頭）は[begin,begin)にマッチしない」
if( regex_match( str.begin(), str.end(), sre, regex_constants::match_not_bol ) )
{
    // ここには絶対にこない！
}
```

regex_match()の使い方に関する完全なプログラム例はここにある。利用可能な多重定義の完全なリストは regex_match()のリファレンスを見よ。

部分文字列のマッチを検索する

入力シーケンスに正規表現にマッチする部分シーケンスが含まれているか調べるには regex_search()を使用する regex_search()は入力シーケンスの先頭で正規表現マッチを試行し、マッチを見つけるかシーケンスの終端に到達するまでシーケンスを走査する。

その他のすべての面で regex_search()の動作は regex_match()と似たようなものである（上を見よ）。std::string、C 形式の null 終端文字列、イテレータの範囲といった双方向範囲を取り扱うという点が特にそうである。正規表現のイテレータ型と入力シーケンスの型を一致させなければならない、ということについても同様の注意が必要である。regex_match()と同様、match_results<>構造体を与えて検索結果を受け取ったり、match_flag_type ビットマスクを使ってマッチをどのように行うかを制御できる。

regex_search()の使い方に関する完全なプログラム例はここにある。利用可能な多重定義の完全なリストは regex_search()のリファレンスを見よ。

結果へのアクセス

概要

regex_match() および regex_search() の成否が分かるだけでは十分でない場合もある。regex_match()、regex_search()に match_results<>型のオブジェクトを渡すと、アルゴリズムが完全に成功した後 match_results<>に、正規表現のどの部分がシーケンスのどの部分にマッチしたかの追加情報が入る。Perl ではこれらの部分シーケンスを**後方参照**といい、変数 $1、$2、...に格納される。xpressive では sub_match<>型のオブジェクトであり、match_results<>構造体に格納される。これらは sub_match<>オブジェクトのベクタとして振舞う。

match_results

さて、正規表現アルゴリズムに match_results<>オブジェクトを渡し、アルゴリズムが成功したとする。結果を調べたくなるところだ。match_results<>オブジェクトを使ってすることといえば、その内部に格納されている sub_match<>オブジェクトへ添字を介してアクセスすることがほとんどである。しかし match_results<>オブジェクトには他にも少し使い道がある。

what という名前の match_results<>オブジェクトに格納されている情報にアクセスする方法を以下の表に示す。

表 5: match_results<>のアクセス子

アクセス子	効果
what.size()	部分マッチの総数を返す。マッチ全体は 0 番目の部分マッチとして格納されるため、アルゴリズムが成功した場合は結果は常に 0 より大きい。
what[n]	*n* 番目の部分マッチを返す。
what.length(n)	*n* 番目の部分マッチの長さを返す。what[n].length()と同じ。
what.position(n)	*n* 番目の部分マッチ開始位置の入力シーケンス内におけるオフセットを返す。
what.str(n)	*n* 番目の部分マッチから構築した std::basic_string<>を返す。what[n].str()と同じ。
what.prefix()	入力シーケンスの先頭から全体マッチ先頭までの部分シーケンスを表す sub_match<>オブジェクトを返す。
what.suffix()	全体マッチの終端から入力シーケンスの終端までの部分シーケンスを表す sub_match<>オブジェクトを返す。
what.regex_id()	この match_results<>オブジェクトで最後に使用した basic_regex<>オブジェクトの regex_id を返す。

match_results<>オブジェクトには他にも使い道があるが、文法と入れ子マッチの項であらためて述べることにする。

sub_match

match_results<>オブジェクトに添字を介してアクセスすると sub_match<>オブジェクトが得られる。sub_match<>は基本的にはイテレータの組である。定義は以下のようになっている。

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

std::pair<>を公開継承しているため、sub_match<>は BidirectionalIterator 型の first および second データメンバをもつ。これらは、この sub_match<>が表す部分シーケンスの先頭と終端である。また sub_match<>は論理型の matched データメンバをもち、この sub_match<>が完全マッチに関与する場合に真となる。

名前を sub とした場合の、sub_match<>オブジェクトに格納されている情報にアクセスする方法を以下の表に示す。

表 6: sub_match<>アクセス子

アクセス子	効果
sub.length()	部分マッチの長さを返す。std::distance(sub.first,sub.second)と同じ。
sub.str()	部分マッチから構築した std::basic_string<> を返す。std::basic<char_type>(sub.first,sub.second)と同じ。
sub.compare(str)	部分マッチと str の文字列比較を行う。str は std::basic_string<>、C 形式の null 終端文字列、別の部分マッチのいずれでもよい。sub.str().compare(str)と同じ。

注意！ 結果の無効化

結果は入力シーケンス内のイテレータとして格納される。入力シーケンスが無効になるとマッチ結果もまた無効となる。例えば std::string オブジェクトに対してマッチを行った場合、結果が有効なのは、次にその std::string オブジェクトの非 const メンバ関数を呼び出すまでの間だけである。それ以降は match_results<>オブジェクトに格納されている結果は無効となるため、使用してはならない。

文字列の置換

正規表現が威力を発揮するのはテキスト検索のときだけではない。テキストの**操作**においても有効である。最もありふれたテキスト操作の 1 つが、「検索して置換」である。xpressive は検索と置換のために regex_replace()アルゴリズムを提供する。

regex_replace()

regex_replace()を用いた「検索して置換」処理は簡単である。必要なのは入力シーケンス、正規表現オブジェクト、および書式化文字列か書式化オブジェクトだけである。regex_replace()には複数のバージョンがあり、入力シーケンスを std::string のような双方向コンテナとして受け付けて結果を同じ型の新しいコンテナで返すものや、入力を null 終端文字列で受け付けて std::string を返すもの、イテレータの組で受け付けて結果を出力イテレータに書き込むものがある。置換は書式化シーケンスを含む文字列か書式化オブジェクトで指定する。文字列ベースの置換について、単純な使用例を以下に示す。

```
std::string input("This is his face");
sregex re = as_xpr("his");                  // "his" をすべて検索し、...
std::string format("her");                  // ... "her" で置換する

// regex_replace()の対文字列版を使用
std::string output = regex_replace( input, re, format );
std::cout << output << '\n';

// regex_replace()の対イテレータ版を使用
std::ostream_iterator< char > out_iter( std::cout );
regex_replace( out_iter, input.begin(), input.end(), re, format );
```

上のプログラムは以下を印字する。

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

“his” が**すべて** “her” に置換されることに注意していただきたい。

regex_replace()の使い方に関する完全なプログラム例はここにある。利用可能な多重定義の完全なリストは regex_replace()のリファレンスを見よ。

置換のオプション

regex_replace()アルゴリズムは、省略可能なビットマスク引数により書式化を制御する。使用可能なビットマスク値を以下に示す。

表 7: 書式化フラグ

フラグ	意味
format_default	ECMA-262 の書式化シーケンスを使用する（後述）。
format_first_only	すべてのマッチの中で最初のものだけを置換する。
format_no_copy	入力シーケンス内の、正規表現にマッチしなかった部分を出力シーケンスにコピーしない。
format_literal	書式化文字列をリテラル（即値）として扱う。エスケープシーケンスを一切解釈しなくなる。
format_perl	Perl の書式化シーケンスを使用する（後述）。
format_sed	sed の書式化シーケンスを使用する（後述）。
format_all	Perl の書式化シーケンス、および Boost 固有の書式化シーケンスを使用する。

これらのフラグは xpressive::regex_constants 名前空間内にある。置換の引数が文字列ではなく関数オブジェクトである場合は、format_literal、format_perl、format_sed および format_all は無視される。

ECMA-262 書式化シーケンス

上記のフラグを指定せずに書式化文字列を渡した場合は、ECMAScript の標準である ECMA-262 の定義が使われる。ECMA-262 モードで使用するエスケープシーケンスを以下に示す。

表 8: 書式化エスケープシーケンス

エスケープシーケンス	意味
$1、$2、...	部分マッチ
$&	マッチ全体
$`	マッチの前
$'	マッチの後
$$	リテラルの文字'$'

その他、$で始まるシーケンスは、単純にそれ自身を表す。例えば書式化文字列が $a であれば、出力シーケンスに “$a” が挿入される。

sed 書式化シーケンス

regex_replace() に format_sed フラグを指定した場合に使用するエスケープシーケンスを以下に示す。

表 9: sed 書式化エスケープシーケンス

エスケープシーケンス	意味
\1、\2、...	部分マッチ
&	マッチ全体
\a	リテラルの'\a'
\e	リテラルの char_type(27)
\f	リテラルの'\f'
\n	リテラルの'\n'

\r	リテラルの'\r'
\t	リテラルの'\t'
\v	リテラルの'\v'
\xFF	リテラルの char_type(0xFF)。*F* は 16 進数字
\x{FFFF}	リテラルの char_type(0xFFFF)。*F* は 16 進数字
\cX	制御文字 *X*

Perl 書式化シーケンス

regex_replace()に format_perl フラグを指定した場合に使用するエスケープシーケンスを以下に示す。

表 10: Perl 書式化エスケープシーケンス

エスケープシーケンス	意味
$1、$2、...	部分マッチ
$&	マッチ全体
$`	マッチの前
$'	マッチの後
$$	リテラルの'$'文字
\a	リテラルの'\a'
\e	リテラルの char_type(27)
\f	リテラルの'\f'
\n	リテラルの'\n'
\r	リテラルの'\r'
\t	リテラルの'\t'
\v	リテラルの'\v'
\xFF	リテラルの char_type(0xFF)。*F* は 16 進数字
\x{FFFF}	リテラルの char_type(0xFFFF)。*F* は 16 進数字
\cX	制御文字 *X*
\l	次の文字を小文字にする
\L	次に\E が現れるまで残りの置換を小文字にする
\u	次の文字を大文字にする
\U	次に\E が現れるまで残りの置換を大文字にする
\E	\L、\U の効果を終了する
\1、\2、...	部分マッチ
\g<name>	名前付き後方参照 *name*

Boost 固有の書式化シーケンス

regex_replace()に format_all を指定した場合に使用するエスケープシーケンスは上に挙げた format_perl と同じである。さらに以下の形式の条件式を使用する。

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

*N*は部分マッチを表す 10 進数字である。この部分マッチがマッチ全体に含まれる場合は置換は *true-expression* となり、それ以外の場合は*false-expression* となる。このモードでは括弧()でグループ化を行う。リテラルの括弧は\(のようにエスケープが必要である。

書式化オブジェクト

テキスト置換において、書式化文字列の表現能力が常に十分とは限らない。入力文字列を環境変数で置換して出力文字列にコピーする単純な例を考えよう。こういう場合は、書式化**文字列**ではなく書式化**オブジェクト**を使ったほうがよい。次のコードを考えよう。"$(xyz)"の形式で埋め込まれた環境変数を検索し、辞書に照らし合わせて見つかった置換文字列を算出する。

```
#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)\"");

    // "$(XYZ)"のような文字列を検索し、env["XYZ"]の結果で置換する
    sregex envar = "$(" >> (s1 = +_w) >> ')';
    std::string output = regex_replace(input, envar, format_fun);
    std::cout << output << std::endl;

    return 0;
}
```

この場合、関数 format_fun()を使って置換文字列をその場で算出している。この関数は現在のマッチ結果が入った match_results<>オブジェクトを受け取る。format_fun()は「1 番目の部分マッチ」をグローバルな env 辞書のキーに使っている。上記コードは次を表示する。

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

書式化オブジェクトは単純な関数である必要はなく、クラス型のオブジェクトでもよい。また文字列を返す以外に、出力イテレータに置換結果を書き込んでもよい。以下は上記と機能的に等価なコードである。

```
#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;

    return 0;
}
```

書式化オブジェクトは、シグニチャが以下の表に示す 3 種類のどれか 1 つである呼び出し可能オブジェクト（関数か関数オブジェクト）でなければならない。表中の `fmt` は関数ポインタか関数オブジェクト、`what` は `match_results<>`オブジェクト、`out` は OutputIterator、`flags` は `regex_constants::match_flag_type` の値である。

表 11: 書式化オブジェクトのシグニチャ

書式化オブジェクトの呼び出し	戻り値の型	意味
`fmt(what)`	文字の範囲（`std::string` など）か null 終端文字列	正規表現にマッチした文字列を書式化オブジェクトが返した文字列で置換する。
`fmt(what, out)`	OutputIterator	書式化オブジェクトは置換文字列を `out` に書き込み、`out` を返す。

`fmt(what, out, flags)`	OutputIterator	書式化オブジェクトは置換文字列を `out` に書き込み、`out` を返す。`flags` 引数は `regex_replace()` アルゴリズムに渡したマッチフラグの値。

書式化式

書式化**文字列**、書式化**オブジェクト**に加えて、`regex_replace()` は書式化**式**も受け付ける。書式化式は文字列を生成するラムダ式である。使用する構文は後述する意味アクションと同じである。文字列を `regex_replace()` を用いて環境変数で置換する上の例を書式化式を使って書き直すと、次のようになる。

```
#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;

    return 0;
}
```

上のコードの `ref(env)[s1]` が書式化式で、1 番目の部分マッチの値 `s1` を辞書 `env` のキーとするという意味となる。ここで `xpressive::ref()` を使っているのは、ローカル変数 `env` への参照を**遅延**して `s1` の置換対象が判明するまで添字演算を遅らせるためである。

文字列の分割とトークン分割

`regex_token_iterator<>` はテキスト操作の世界における GINSU[6] のナイフである。薄切りもさいの目切りも思いのまま！ 本節では高度に設定可能な `regex_token_iterator<>` で入力シーケンスを分割する方法を述べる。

概要

`regex_token_iterator<>` は入力シーケンス、正規表現、省略可能な設定引数で初期化する。`regex_token_iterator<>` は `regex_search()` を使って、シーケンス内で最初に正規表現にマッチする位置を見つける。`regex_token_iterator<>` を参照はがしすると、`std::basic_string<>` 形式で**トークン**を返す。どの文字列を返すかは設定引数による。既定ではマッチ全体に相当する文字列を返すが、マーク済み部分式のみならずシーケンス内のマッチ**しなかった**部分を返すことも可能である。

6 訳注 刃物メーカー（http://www.genuineginsu..com/）。GINSU のナイフはよく切れると評判らしいです。Wikipedia によるとテレビ CM が画期的なものだったとか。

regex_token_iterator<>をインクリメントすると次のトークンに移動する。次がどのトークンかは設定引数による。単純に現在のマッチにおける異なるマーク済み部分式の場合もあれば、次のマッチの全体か一部分である場合、マッチ**しなかった**部分である場合もある。

以上のことからわかるように、regex_token_iterator<>には多くの機能がある。すべてを説明するのは難しいが、いくつか例を見れば理解できるだろう。

例 1: 単純なトークン分割

この例では regex_token_iterator<>を使ってシーケンスを単語のトークンに切っている。

```
std::string input("This is his face");
sregex re = +_w;                        // 単語を検索する

// 入力中の単語をすべて走査する
sregex_token_iterator begin( input.begin(), input.end(), re ), end;

// すべての単語を std::cout に出力する
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

このプログラムは以下を表示する。

```
This
is
his
face
```

例 2: 単純なトークン分割・リローデッド

この例も regex_token_iterator<>を使ってシーケンスを単語トークンに切っているが、正規表現を区切りとして使っている。regex_token_iterator<>コンストラクタの最後の引数に-1 を渡すと、入力内の正規表現にマッチ**しなかった**部分がトークンとなる。

```
std::string input("This is his face");
sregex re = +_s;                        // 空白を検索する

// 入力中の非空白をすべて走査する。-1 に注意
sregex_token_iterator begin( input.begin(), input.end(), re, -1 ), end;

// すべての単語を std::cout に出力する
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

このプログラムは以下を出力する。

```
This
is
```

```
his
face
```

例 3: 単純なトークン分割・レボリューションズ[7]

この例も regex_token_iterator<>を使って日付の束が入ったシーケンスを年だけのトークンに切っている。regex_token_iterator<>コンストラクタの最後の引数に正の整数 N を渡すと、各マッチの *N* 番目のマーク済み部分式のみがトークンとなる。

```
std::string input("01/02/2003 blahblah 04/23/1999 blahblah 11/13/1981");
sregex re = sregex::compile("(\\d{2})/(\\d{2})/(\\d{4})"); // 日付を検索する

// 入力中のすべての年を走査をする。3（3 番目の部分式）に注意
sregex_token_iterator begin( input.begin(), input.end(), re, 3 ), end;

// すべての単語を std::cout に出力する
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

このプログラムは以下を出力する。

```
2003
1999
1981
```

例 4: あまり単純でないトークン分割

この例は 1 つ前のものと似ているが、年だけでなく月と日をトークンに入れている点が異なる。regex_token_iterator<>コンストラクタの最後の引数に整数の配列{I,J,...}を渡すと、各マッチの *I* 番目、*J* 番目、...のマーク済み部分式がトークンとなる。

```
std::string input("01/02/2003 blahblah 04/23/1999 blahblah 11/13/1981");
sregex re = sregex::compile("(\\d{2})/(\\d{2})/(\\d{4})"); // 日付を検索する

// 入力中の年月日を走査する
int const sub_matches[] = { 2, 1, 3 }; // 日、月、年
sregex_token_iterator begin( input.begin(), input.end(), re, sub_matches ), end;

// すべての単語を std::cout に出力する
std::ostream_iterator< std::string > out_iter( std::cout, "\n" );
std::copy( begin, end, out_iter );
```

このプログラムは以下を出力する。

```
02
01
2003
```

7 訳注 マトリックスですね。

```
23
04
1999
13
11
1981
```

sub_matches 配列により、regex_token_iterator<>は最初に 2 番目の部分マッチ、次に 1 番目の部分マッチ、最後に 3 番目の部分マッチの値を取る。イテレータをインクリメントすると regex_search() を使って次のマッチを検索する。ここで処理が繰り返され、イテレータは 2 番目の部分マッチを取り、次に 1 番目...となる。

名前付き捕捉

概要

正規表現が複雑になると、番号付き捕捉を取り扱うのが苦痛になる場合がある。左括弧の数を数えてどの捕捉に対応しているのか調べるのはつまらない仕事である。さらに面白くないのは、正規表現を編集するだけで捕捉に新しい番号が割り振られて古い番号を使っていた後方参照が無効になることである。

他の正規表現エンジンでは、**名前付き捕捉**という機能でこの問題を解決している。この機能を使うと捕捉に名前を付けることができ、番号ではなく名前で捕捉を後方参照できる。xpressive も動的・静的正規表現の両方で名前付き捕捉をサポートする。

動的名前付き捕捉

動的正規表現については、xpressive は他の一般的な正規表現エンジンの名前付き捕捉の構文に従う。`(?P<xxx>...)` で名前つき捕捉を作成し、`(?P=xxx)` でこの捕捉を後方参照する。名前付き後方参照を作成し後方参照する例を以下に示す。

```
// 1 文字にマッチする"char"という名前付き捕捉を作成し、名前により後方参照する。
sregex rx = sregex::compile("(?P<char>.)(?P=char)");
```

上の正規表現は同じ文字が 2 つ続いた部分を検索する。

名前付き捕捉を使ってマッチか検索を行った後、捕捉の名前を使って match_results<> により名前付き捕捉にアクセスする。

```
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;
}
```

上のコードは以下を表示する。

```
char = e
```

名前付き捕捉を置換文字列から後方参照することも可能である。`\g<xxx>` という構文である。文字列置換において名前付き捕捉

を使用する例を以下に示す。

```
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;
```

名前付き捕捉を使用するには format_perl を指定しなければならないことに注意していただきたい。\g<xxx>構文を解釈するのは Perl の構文だけである。上のコードは以下を表示する。

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

静的名前付き捕捉

静的正規表現を使う場合は、名前付き捕捉の作成と使用はより簡単である。mark_tag 型を使って s1、s2 のような変数を作成するが、より意味のある名前を与えることができる。静的表現を使うと上の例は以下のようになる。[8]

```
mark_tag char_(1); // char_ は s1 の別名となる
sregex rx = (char_= _) >> char_;
```

マッチを行った後、mark_tag を match_results<>の添字にして名前付き捕捉にアクセスする。

```
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;
}
```

上のコードは以下を表示する。

```
char = e
```

regex_replace()を使って文字列置換を行う場合、以下のように名前付き捕捉を使用して**書式化式**を作成できる。

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

上のコードは以下を表示する。

8　訳注　リファレンスの項にあるとおり、mark_tag の初期化に使用する整数は正規表現内で一意でなければなりません。

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

注意

書式化式を使用するには<boost/xpressive/regex_actions.hpp>をインクルードしなければならない。

文法と入れ子マッチ

概要

正規表現をC++の式で表現することの重要な利点の1つは、正規表現中から他のC++コードやデータに容易にアクセスできることである。これにより、他の正規表現で不可能なプログラミングイディオムが可能になる。特に注意していただきたいのは、正規表現が他の正規表現を参照する機能で、これにより正規表現の外部で文法を構築できる。この節では正規表現を他の正規表現に値や参照で組み込む方法、正規表現が他の正規表現を参照したときの振る舞い、解析が成功した後の結果木にアクセスする方法を説明する。

値による正規表現の組み込み

basic_regex<>オブジェクトは値のセマンティクスをもつ。正規表現オブジェクトが別の正規表現定義の右辺に現れると、値による組み込みが起こるとみなされる。つまり、入れ子の正規表現のコピーが外側の正規表現に格納される。内側の正規表現は、パターンマッチ時に外側の正規表現により呼び出される。内側の正規表現をマッチに対して完全に消耗すると、マッチを成功させるためにバックトラックが起こる。

単語単位の正規表現検索機能をもつテキストエディタを考える。これをxpressiveで実装すると次のようになる。

```
find_dialog dlg;
if( dialog_ok == dlg.do_modal() )
{
    std::string pattern = dlg.get_text();           // ユーザーが入力したパターン
    bool whole_word = dlg.whole_word.is_checked(); // ユーザーが単語単位のオプションを選択したか？

    sregex re = sregex::compile( pattern );         // パターンのコンパイル

    if( whole_word )
    {
        // 正規表現を単語の先頭、単語の終端表明で囲む
        re = bow >> re >> eow;
    }

    // ... reを使う ...
}
```

この行に注目する。

```
// 正規表現を単語の先頭、単語の終端表明で囲む
re = bow >> re >> eow;
```

この行は既存の正規表現を値で組み込んだ正規表現を新たに作成し、元の正規表現に代入している。元の正規表現のコピーが右辺にあるので、これは期待したとおりに動作する。つまり、新しい正規表現の振る舞いは元の正規表現を単語先頭と単語終端の表明で囲んだものとなる。

注意

既定では正規表現オブジェクトは値で組み込まれるため、`re = bow >> re >> eow` は再帰正規表現を定義**しない**ことに注意していただきたい。次の節では、正規表現を参照で組み込んで再帰正規表現を定義する方法を述べる。

参照による正規表現の組み込み

再帰正規表現および文脈自由文法を構築するには、値による正規表現の組み込みでは不十分である。正規表現を自己参照的にする必要がある。大半の正規表現エンジンにはそういった能力はないが、xpressive では可能である。

ヒント

理論コンピュータ科学者は、自己参照的な正規表現は「正規（正則）」ではないと指摘するかもしれない。そういう意味では、厳密には xpressive は本当は**正規**表現エンジンではない。しかし Larry Wall がかつてこう言ったことがある。「項 [regular expression] は我々のパターンマッチエンジンとともに成長した。言語の必要性と戦うつもりはない。」

次のコードを考える。`by_ref()` ヘルパを使って、数の合った入れ子の括弧にマッチする再帰正規表現を定義している。

```
sregex parentheses;
parentheses                         // 数の合った括弧群は...
    = '('                           // 最初に１つの開き括弧があり...
        >>                          // その後ろに...
         *(                         // ０か１つ以上の..
            keep( +~(set='(',')') ) // 括弧以外のものの塊か...
          |                         // あるいは...
            by_ref(parentheses)     // 数の合った括弧群があり
          )                         //    （これだ、再帰している！）...
        >>                          // その後ろに...
      ')'                           // １つの閉じ括弧がある
    ;
```

数の合った入れ子のタグに対するマッチは重要なテキスト処理であり、「旧式の」正規表現では不可能なことの 1 つである。`by_ref()` ヘルパがこれを可能にする。これによりある正規表現を別の正規表現から**参照により**組み込むことができる。右辺が `parentheses` を参照で保持しているので、`parentheses` に右辺を代入すると循環が生まれ再帰的に実行される。

文法の構築

正規表現が自己再帰的になりさえすれば、もう後戻りする必要はない。楽しみにしていたことがすべて可能になる。特に正規表現の外部で文法を構築できるようになる。text-book 文法の例を見よう。ちょっとした計算機だ。

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

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

上で定義した正規表現 `expression` は正規表現としては非常に注目すべき動作をする。数式にマッチするのである。例えば入力文字列が`"foo 9*(10+3) bar"`であれば、このパターンは`"9*(10+3)"`にマッチする。この正規表現がマッチするのは正しい形式の数式、つまり括弧の数が合っており、中置演算子が引数を 2 つもつ場合のみである。他の正規表現エンジンでこれを試してはいけませんぞ！

この正規表現文法をもっとよく見てみよう。循環していることに注意していただきたい。`expression` は `term` を使って実装してあり、`term` は `factor` を使って実装してある。`factor` は `group` を使って実装してあり、`group` は `expression` を使って実装してある。というわけでループが閉じている。大抵の場合、循環文法の定義は正規表現オブジェクトの前方宣言とこれら未初期化の正規表現の参照による組み込みにより行う。上の文法では、未初期化の正規表現オブジェクトを参照する必要があるのは 1 箇所だけである。それが `group` の定義であり、`by_ref()` を使って `expression` を参照により組み込んでいる。他の正規表現オブジェクトはすべて初期化済みで値が変化することもないため、値による組み込みで事足りている。

ヒント：可能な限り、値による組み込みを使え

通常、正規表現の組み込みは参照よりも値で行うほうが望ましい。そのほうが分かりやすいし、パターンマッチが少し高速になる。その上、値のセマンティクスは簡単で文法の推論が容易になる。正規表現の「コピー」の負荷については心配しないでいただきたい。各正規表現オブジェクトはコピー間で実装を共有する。

動的正規表現文法

`regex_compiler<>`を使用して動的正規表現の外部で文法を構築することもできる。名前付きの正規表現を作成し、他の正規表現から名前で参照するのである。各 `regex_compiler<>`インスタンスは名前と正規表現の対応を保持する。

名前付き動的正規表現を作成するには、正規表現の先頭に`(?$name=)`を付ける。*name* は正規表現の名前である。名前付き正規表現を他の正規表現から名前で参照するには`(?$name)`とする。名前付き正規表現は他の正規表現から参照する時点では存在していなくても構わないが、正規表現を使用する時点では存在していなければならない。

以下のコード片は、動的正規表現文法を使って上の計算機の例を実装している。

```
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))
{
     // "9*(10+3)" を印字する:
     std::cout << what[0] << std::endl;
}
```

静的正規表現の場合と同様、入れ子の正規表現を呼び出すと入れ子のマッチ結果が作成される（以下の**入れ子の結果**を見よ）。結果はマッチした文字列の完全な解析木である。静的正規表現と異なり、動的正規表現は常に値ではなく参照による組み込みとなる。

循環パターンにコピーにメモリ管理まで、まあ何てこと！

上の計算機の例で非常に複雑なメモリ管理の問題が持ち上がる。4 つの正規表現オブジェクトは直接・間接的に、また値・参照でお互いを参照している。このうちの 1 つを関数から返し、残りがスコープの外に出るとどうなるのか？ 参照はどうなるのか？ 答えは、正規表現オブジェクトは内部に参照カウントを持つため必要な限り正規表現による参照は保持される、である。よって正規表現オブジェクトを値で渡しても、それがスコープの外に行ってしまった正規表現オブジェクトを参照していたとしても問題は起きない。

参照カウントに詳しい人はおそらくその唯一の弱点についてもご存知と思う。循環参照である。正規表現オブジェクトを参照カウントすると、計算機の例で作成したような循環はどうなるのか？ リークが起こるのか？ 答えはノーであり、リークは起きない。basic_regex<>オブジェクトは技巧的な参照追跡コードを使っており、最後の外部参照が無くなったときに循環正規表現文法はクリーンアップされる。そういうわけで心配無用だ。好きなだけ循環文法を作成したり、正規表現オブジェクトを渡したりコピーしていただきたい。高速かつ高効率で、リークや懸垂参照（dangling references）が起きないことが保証されている。

入れ子の正規表現と部分マッチのスコープ

正規表現を入れ子にすると部分マッチのスコープの問題が持ち上がる。内側と外側の両方の正規表現が同じ部分マッチのベクタを読み書きすると、混乱が起こる。外側の正規表現が書き込んだ部分マッチを内側の正規表現が台無しにするわけだ。例えば、これはどうなるか。

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

外側の正規表現が書き込んだ部分マッチを内側の正規表現が上書きしているが、おそらくこのコードの作者が意図するところで

はないだろう。内側の正規表現がユーザーから入力である場合は、特に大問題である。内側の正規表現が部分マッチのベクタを破壊するかどうか知る方法が無いのである。これは明らかに許容できるものではない。

代わりにどうするのかというと、入れ子の正規表現を呼び出すたびに自身のスコープを形成する。つまり入れ子の正規表現はそれぞれ対象となる部分マッチのベクタについて自分用のコピーを取得するため、外側の正規表現の部分マッチを内側の正規表現が台無しにする可能性は無くなる。例えば上で定義した正規表現 `outer` は、当然`"ABBA"`にマッチする。

入れ子の結果

入れ子の正規表現が自身の部分マッチをもつのであれば、マッチ成功後にそれらにアクセスする方法があってしかるべきである。`regex_match()` か `regex_search()` の後、`match_results<>` 構造体は入れ子の結果を表す木の頂点のように振舞う。`match_results<>` クラスは、入れ子の正規表現の結果を表す `match_results<>` 構造体の順序付きシーケンスを返す `nested_results()` メンバ関数を提供する。入れ子の結果の順序は、入れ子の正規表現がマッチした順序と同じである。

前に見た、数の合った入れ子の括弧の正規表現を例にとる。

```
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 ) )
{
    // マッチ全体を表示する
    std::cout << what[0] << '\n';

    // 入れ子の結果を表示する
    std::for_each(
        what.nested_results().begin(),
        what.nested_results().end(),
        output_nested_results() );
}
```

このプログラムは以下を表示する。

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

結果がどのように入れ子になるか、それらが見つかった順に格納されていることが分かったと思う。

ヒント

例の節にある output_nested_results の定義を見よ。

入れ子の結果のフィルタリング

1 つの正規表現の中に複数の入れ子の正規表現があり、どの結果がどの正規表現に対応するのか知りたい場合がある。basic_regex<>::regex_id()とmatch_results<>::regex_id()が役に立つ場面である。入れ子の結果を走査しているときに、結果の正規表現 ID と目的の正規表現オブジェクトの ID を比較するとよい。

これを少し容易にするために、xpressive は特定の入れ子正規表現に相当する結果だけを列挙する述語を提供している。これが regex_id_filter_predicate であり、Boost.Iterator とともに使用することを意図している。以下のように使用する。

```
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();

    // 名前（name）か整数（integer）だけを選択する述語フィルタを宣言する
    sregex_id_filter_predicate name_id( name.regex_id() );
    sregex_id_filter_predicate integer_id( integer.regex_id() );

    // 正規表現 name の結果だけを走査する
    std::for_each(
        boost::make_filter_iterator( name_id, begin, end ),
        boost::make_filter_iterator( name_id, end, end ),
        output_result
        );

    std::cout << '\n';

    // 正規表現 integer の結果だけを走査する
    std::for_each(
        boost::make_filter_iterator( integer_id, begin, end ),
        boost::make_filter_iterator( integer_id, end, end ),
        output_result
        );
}
```

ここで output_results は smatch を受け取りマッチ全体を表示する単純な関数である。特定の入れ子正規表現に相当する結果だけを選択するのに regex_id_filter_predicate を basic_regex<>::regex_id() と Boost.Iterator の boost::make_filter_iterator()とともに使っている点に注意していただきたい。このプログラムは以下を表示する。

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

意味アクションとユーザー定義表明

概要

入力文字列を解析し、そこから std::map<>を構築したいとする。このような場合、正規表現では不十分である。正規表現マッチの部分で**何か**をしたい。xpressive は、静的正規表現の部分に意味アクションを結びつける方法を提供する。本節ではその方法を説明する。

意味アクション（Semantic Actions）

以下のコードを考える。xpressive の意味アクションを使って単語と整数の組からなる文字列を解析し、std::map<>に詰め込んでいる。

```
#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");

    // => で区切られた単語と整数にマッチし、
    // 結果を std::map<>に詰め込む
    sregex pair = ( (s1= +_w) >> "=>" >> (s2= +_d) )
        [ ref(result)[s1] = as<int>(s2) ];

    // 空白で区切られた 1 つ以上の単語・整数の組にマッチする。
    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;
}
```

このプログラムは以下を印字する。

```
1
23
456
```

正規表現 pair は、パターンとアクションの 2 つの部分からなる。単語のマッチを 1 番目の部分マッチで捕捉し、"=>"で区切られた整数のマッチを 2 番目の部分マッチで捕捉するというのがパターンが表現するところである。アクションは角括弧 [ref(result)[s1] = as<int>(s2)]の内側である。これは 1 番目の部分マッチを results 辞書の添字に使用し、そこに 2 番目の部分マッチ

を整数に変換した結果を代入するという意味である。

> **注意**
>
> 静的正規表現で意味アクションを使用するには、<boost/xpressive/regex_actions.hpp>をインクルードしなければならない。

このコードはどのように動作するのだろう？ 静的正規表現の残りの部分だけ見ると括弧の間は式テンプレートになっている。これでアクションがコード化され、後で実行される。式 ref(result) は result への遅延参照を作成する。より大きな式である ref(result)[s1]は辞書に対する添字操作の遅延である。後でこのアクションを実行すると s1 は 1 番目の sub_match<>で置換される。同様に as<int>(s2) を実行すると s2 は 2 番目の sub_match<> で置換される。as<>アクションは引数を Boost.Lexical_cast を使って要求の型に変換する。アクション全体の効果としては、新しい単語・整数の組を辞書に挿入する、となる。

> **注意**
>
> <boost/ref.hpp>の関数 boost::ref()と<boost/xpressive/regex_actions.hpp>の boost::xpressive::ref() には重大な違いがある。前者は通常の参照とほぼ同様の振る舞いをする素の reference_wrapper<> を返す。一方 boost::xpressive::ref() が返すのは、遅延実行する式内で使用する**遅延**参照である。これが、result が s1 を受け取る operator[]をもたないにも関わらず ref(result)[s1]とする理由である。

部分マッチのプレースホルダ s1、s2 に加えて、アクションが結び付けられている部分式にマッチした文字列を後方参照するのにアクション内で使用するプレースホルダー _ がある。例えば以下の正規表現は数字列にマッチし、それらを整数として解釈して結果をローカル変数に代入する。

```
int i = 0;
// ここで _ は (+_d)にマッチしたすべての文字を後方参照する
sregex rex = (+_d)[ ref(i) = as<int>(_) ];
```

アクションの遅延実行

アクションを正規表現のある部分に結び付けてマッチを行うとは、実際にはどういう意味なのか？ アクションが実行されるのはいつなのか？ アクションが繰り返し部分式の一部である場合は、アクションが実行される回数は 1 度なのか複数回なのか？ また部分式が最初はマッチしていたが正規表現の残りの部分がマッチせず最終的に失敗した場合は、アクションはまったく実行されないのか？

答えは既定では、アクションは**遅延**実行される、である。部分式が文字列にマッチすると、そのアクションはアクションが参照する部分マッチの現在の値とともに待ち行列に置かれる。マッチアルゴリズムがバックトラックしなければならなくなると、アクションは必要に応じて待ち行列から取り出される。アクションが実際に実行されるのは、正規表現全体のマッチが成功した後だけである。regex_match()が制御を返す直前の段階で、これらは待ち行列に追加した順番で一度にすべて実行される。

例として、以下の数字を見つけるたびにカウンタを増やす正規表現を考える。

```
int i = 0;
std::string str("1!2!3?");
// 感嘆符の付いた数字は数えるが、疑問符付きのものは数えない。
sregex rex = +( _d [ ++ref(i) ] >> '!' );
regex_search(str, rex);
assert( i == 2 );
```

アクション++ref(i)は3回（数字が見つかるたびに1回ずつ）待ち行列に入る。しかし**実行**されるのは2回だけ（後ろに"!"文字がある数字1字について1回ずつ）である。"?"文字に遭遇するとマッチアルゴリズムはバックトラックを行い、待ち行列から最後のアクションを削除する。

アクションの即時実行

意味アクションを即時実行したい場合は、そのアクションを含む部分式を keep() で包む。keep()は当該部分式についてバックトラックを無効にし、その部分式の待ち行列に入っているあらゆるアクションを keep()の終了とともに実行する。これにより、あたかもkeep()内の部分式が別の正規表現オブジェクトにコンパイルされ、keep()のマッチングが regex_search()を個別に呼び出して実行されたかのようになる。結果この部分式は文字にマッチしアクションを実行するが、バックトラックも巻き戻しもしない。例えば上の例を以下のように書き換えたとする。

```
int i = 0;
std::string str("1!2!3?");
// 数字をすべて数える。
sregex rex = +( keep( _d [ ++ref(i) ] ) >> '!' );
regex_search(str, rex);
assert( i == 3 );
```

部分式_d [++ref(i)]を keep()で包んだ。こうすることでこの正規表現が数字にマッチするとアクションが待ち行列に入り、"!"文字のマッチを試行する前に即時実行されるようになる。この場合、アクションは3回実行される。

注意

keep()と同様、before()とafter()内のアクションも、その部分式がマッチしたときに早期実行される。

遅延関数

ここまで変数と演算子からなる意味アクションの記述方法について見てきたが、意味アクションから関数を呼び出す方法についてはどうだろう？ xpressive にはそのための機構がある。

まず関数オブジェクト型を定義する。以下の例は引数に対して push()を呼び出す関数オブジェクトである。

```
struct push_impl
{
    // 戻り値の型（tr1::result_of のために必要）
    typedef void result_type;
```

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

次に xpressive の function<>テンプレートを使って push という名前の関数オブジェクトを定義する。

```
// グローバルな"push"関数オブジェクト。
function<push_impl>::type const push = {{}};
```

初期化はいささか奇妙に見えるが、push を静的に初期化するためである。これは実行時に構築する必要はないということを意味する。以下のように push を意味アクション内で使用する。

```
std::stack<int> ints;
// 数字がマッチしたら int へキャストし、スタックに積む。
sregex rex = (+_d)[push(ref(ints), as<int>(_))];
```

この方法だとメンバ関数の呼び出しがただの関数呼び出しに見えてしまうことに気付くと思う。意味アクションを、よりメンバ関数呼び出しらしく見えるように記述する方法がある。

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

xpressive は->*を認識し、この式を上のコードとまったく同等に扱う。

関数オブジェクトが引数によって戻り値の型を変えなければならない場合は、result_type 型定義の代わりに result<>メンバテンプレートを使用するとよい。std::pair<>か sub_match<>の first メンバを返す first 関数オブジェクトの例である。

```
// 組の第 1 要素を返す関数オブジェクト。
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、first(s1)により s1 が参照する部分マッチの先頭を指すイテレータを得る。
```

```
function<first_impl>::type const first = {{}};
```

ローカル変数を参照する

上の例で見たように、xpressive::ref()を使用するとアクション内からローカル変数を参照できる。この変数は正規表現による参照に保持されるが、これらの参照が懸垂しないよう注意が必要である。例えば以下のコードでは、bad_voodoo()が制御を返すとiに対する参照が懸垂する。

```
sregex bad_voodoo()
{
    int i = 0;
    sregex rex = +( _d [ ++ref(i) ] >> '!' );
    // エラー！ rexはローカル変数を参照により参照しており、
    // bad_voodoo()が制御を返した後に懸垂する。
    return rex;
}
```

意味アクションを記述するときは、すべての参照が懸垂しないよう注意を払わなければならない。1つの方法は変数を、正規表現が値により保持する共有ポインタにすることである。

```
sregex good_voodoo(boost::shared_ptr<int> pi)
{
    // val()を使ってshared_ptrを値で保持する:
    sregex rex = +( _d [ ++*val(pi) ] >> '!' );
    // OK、rexは整数への参照カウントを保持する。
    return rex;
}
```

上のコードでは、xpressive::val()を使って共有ポインタを値で保持している。アクション内のローカル変数は既定では値で保持されるため、通常この処理は必要ないが、この場合は必要である。アクションを++*piと記述してしまうと即時実行されてしまう。これは++*piが式テンプレートでないためである（++*val(pi)は式テンプレートである）。

アクション内の変数をすべてref()とval()で包むのはうんざりするかもしれない。これを容易にするためにxpressiveはreference<>およびvalue<>テンプレートを提供している。対応を以下の表に示す。

表 12: reference<>とvalue<>

これは...	...以下と等価である

`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 ] >> '!' );`

上で見たようにreference<>を使用する場合、始めにローカル変数を宣言してからreference<>する。local<>を使用するとこの2段階を1つにまとめられる。

表 13: local<>対 reference<>

これは...	...以下と等価である

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

上の例を `local<>`を使用して書き直すと以下のようになる。

```
local<int> i(0);
std::string str("1!2!3?");
// 感嘆符の付いた数字は数えるが、疑問符付きのものは数えない。
sregex rex = +( _d [ ++i ] >> '!' );
regex_search(str, rex);
assert( i.get() == 2 );
```

`local<>::get()`を使ってローカル変数の値にアクセスしていることに注意していただきたい。また `reference<>`同様、`local<>`が懸垂参照を作成する可能性があることに注意が必要である。

非ローカル変数を参照する

この節の最初で、正規表現を使って単語・整数の組からなる文字列を解析して `std::map<>`に詰め込む例を見た。この例では辞書と正規表現を定義しておき、いずれかがスコープから出る前にそれらを使う必要があった。正規表現を先に定義しておき、異なる複数の辞書に書き込みたい場合はどうすればよいだろうか？ 正規表現オブジェクトに辞書に対する参照を直接組み込むのではなく、`regex_match()`アルゴリズムに辞書を渡すようにしてはどうか。プレースホルダを定義し、意味アクション内で辞書そのものの代わりに使用する。後でいずれかの正規表現アルゴリズムを呼び出すときに実際の辞書オブジェクトへ参照を束縛できる。以下のようにする。

```
// 辞書オブジェクトのプレースホルダを定義する:
placeholder<std::map<std::string, int> > _map;

// => で区切られた単語と整数にマッチし、
// 結果を std::map<>に詰め込む
sregex pair = ( (s1= +_w) >> "=>" >> (s2= +_d) )
    [ _map[s1] = as<int>(s2) ];

// 空白で区切られた 1 つ以上の単語・整数の組にマッチする。
sregex rx = pair >> *(+_s >> pair);

// 解析する文字列
std::string str("aaa=>1 bbb=>23 ccc=>456");

// 結果を書き込む実際の辞書:
std::map<std::string, int> result;

// _map プレースホルダを実際の辞書に束縛する
smatch what;
what.let( _map = result );

// マッチを実行し結果の辞書に書き込む
```

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

このプログラムは以下を表示する。

```
1
23
456
```

placeholder<>を使って_mapを定義しており、これがstd::map<>変数の代理となる。意味アクション内でこのプレースホルダを辞書として使用できる。次にmatch_results<>構造体を定義してwhat.let(_map = result);で実際の辞書をプレースホルダに束縛する。regex_match()呼び出しは、意味アクション内のプレースホルダをresultへの参照で置換したかのように振舞う。

注意

意味アクション内のプレースホルダは**実際には**実行時に変数への参照で置換されない。正規表現オブジェクトはいずれの正規表現アルゴリズムでも変更されることはないので、複数のスレッドで使用しても安全である。

regex_iterator<>かregex_token_iterator<>を使用する場合は、遅延束縛されたアクションの引数は少し異なる。正規表現イテレータのコンストラクタは、引数の束縛を指定する引数を受け付ける。変数をそのプレースホルダに束縛するのに使用するlet()関数がある。以下のコードに方法を示す。

```
// 辞書オブジェクトのプレースホルダを定義する:
placeholder<std::map<std::string, int> > _map;

// => で区切られた単語と整数にマッチ
sregex pair = ( (s1= +_w) >> "=>" >> (s2= +_d) )
    [ _map[s1] = as<int>(s2) ];

// 解析する文字列
std::string str("aaa=>1 bbb=>23 ccc=>456");

// 結果を書き込む実際の辞書:
std::map<std::string, int> result;

// regex_iteratorを作成し、すべてのマッチを検索する
sregex_iterator it(str.begin(), str.end(), pair, let(_map=result));
sregex_iterator end;

// すべてのマッチについて結果の辞書に書き込む
while(it != end)
    ++it;

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

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

このプログラムは以下を出力する。

```
1
23
456
```

ユーザー定義表明

正規表現の**表明**については慣れたものだろう。Perl だと表明の例として`^`や`$`があり、それぞれ文字列の先頭・終端にマッチする。xpressive では新たに表明を定義できる。カスタム表明は、マッチの成否を判断する時点で真でなければならない条件である。カスタム表明をチェックするには xpressive の `check()` 関数を使用する。

カスタム表明を定義する方法はいくつかある。一番簡単なのは関数オブジェクトを使うことである。長さが 3 文字か 6 文字のいずれかである部分文字列にマッチする部分式が必要であるとする。そのような述語を以下の構造体で定義する。

```
// 部分マッチが長さ 3 文字か 6 文字であれば真となる述語。
struct three_or_six
{
    bool operator()(sub_match const &sub) const
    {
        return sub.length() == 3 || sub.length() == 6;
    }
};
```

この述語を正規表現で使うには以下のようにする。

```
// 3 文字か 6 文字の単語にマッチする。
sregex rx = (bow >> +_w >> eow)[ check(three_or_six()) ] ;
```

上の正規表現は長さが 3 文字か 6 文字の単語全体にマッチする。述語 `three_or_six` は、カスタム表明が結び付けられた部分式にマッチした部分を後方参照する `sub_match<>` を受け取る。

注意

カスタム表明はマッチの成否に関与する。遅延実行されるアクションとは異なり、カスタム表明は正規表現エンジンがマッチを検索するときに即時実行される。

カスタム表明は意味アクションと同じ構文を用いてインライン定義することもできる。以下は同じカスタム表明をインラインで書き直したものである。

```
// 3 文字か 6 文字の単語にマッチする。
sregex rx = (bow >> +_w >> eow)[ check(length(_)==3 || length(_)==6) ] ;
```

上記において、length()は引数の length()メンバ関数を呼び出す遅延関数であり、_は sub_match を受け取るプレースホルダである。

カスタム表明のインライン記述は、コツが分かってしまえば非常に強力である。（あまり厳密でない意味での）正しい日付にのみマッチする正規表現を書いてみよう。

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

mark_tag month(1), day(2);
// 「月/日/年」形式の正しい日付を検索する。
sregex date =
    (
        // 月は1以上12以下でなければならない
        (month= _d >> !_d)      [ check(as<int>(_) >= 1
                                    && as<int>(_) <= 12) ]
    >>  '/'
        // 日は1以上31以下でなければならない
    >>  (day=   _d >> !_d)      [ check(as<int>(_) >= 1
                                    && as<int>(_) <= 31) ]
    >>  '/'
        // 年は1970以上2038以下とする
    >>  (_d >> _d >> _d >> _d) [ check(as<int>(_) >= 1970
                                    && as<int>(_) <= 2038) ]
    )
    // 月ごとの実際の日数を確認する！
    [ 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;
}
```

このプログラムは以下を印字する。

```
2/28/2006
```

インラインのカスタム表明を使って年・月・日の値の範囲チェックを行っていることに注意していただきたい。"99/99/9999"や"2/30/2006"は正しい日付ではないため、この正規表現はマッチしない（99の月は存在しないし、2月には30日はない）。

記号表と属性

概要

xpressive の正規表現で記号表を構築するには、std::map<>を使うだけでよい。辞書のキーはマッチした文字列であり、辞書の値は意味アクションが返すデータである。xpressive の属性 a1、a2、...、a9 はマッチしたキーに相当する値を保持し、意味アクション内で使用する。記号が見つからなかった場合の属性の既定値を指定することも可能である。

記号表

xpressive の記号表は単純に std::map<>であり、キーは文字列型、値は何でもよい。例えば以下の正規表現は、map1 のキーにマッチし対応する値を属性 a1 に代入する。次に意味アクションにおいて、属性 a1 に格納した値を結果の整数に代入する。

```
int result;
std::map<std::string, int> map1;
// ...（辞書を埋める）
sregex rx = ( a1 = map1 ) [ ref(result) = a1 ];
```

次のコード例は数値の名前を整数に変換する。説明は以下に示す。

```
#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;
    // number_map の文字列でマッチを行い
    // 整数値を'result'に格納する
    // 見つからなければ-1 を'result'に格納する
    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;
}
```

このプログラムは以下を印字する。

```
3
2
-1
```

このプログラムは始めに、数の名前をキー文字列とし対応する整数を値とする数値の辞書を構築している。次に記号表の探索結果を表す属性 a1 を使って静的正規表現を構築している。意味アクション内では属性を整数変数 result に代入している。記号が見つからなければ既定値の-1 を result に代入する。記号が見つからなくてもマッチが成功するために、ワイルドカード*_を使っている。

この例のより完全版は libs/xpressive/example/numbers.cpp にある[9]。このコードは「999,999,999」以下の数の名前（「ダース」のような特殊な数の名前が混ざっていてもよい）を数値に変換する。

記号表のマッチは既定では大文字小文字を区別するが、式を icase() で囲むことにより大文字小文字を区別しないようにできる。

属性

1 つの正規表現内で使用できる属性は最大 9 つであり、a1、a2、...、a9 という名前で boost::xpressive 名前空間内にある。属性の型は代入元の辞書の 2 番目の要素と同じである。属性の既定値は意味アクション内で (a1 | *default-value*) のような構文で指定する。

属性のスコープは適切に設定されるため、((a1=sym1) >> (a1=sym2)[ref(x)=a1])[ref(y)=a1] のようなとてつもないこともできる。内側の意味アクションは内側の a1 を参照し、外側の意味アクションは外側の属性を参照する。これらは型が異なっていてもよい。

> **注意**
>
> xpressive は検索を高速化するために、辞書から不可視の 3 分探索木を構築する。BOOST_DISABLE_THREADS を定義した場合、この不可視の 3 分木は検索後に毎回自身を再構築し、前回の検索頻度に基づいて次回の検索効率を向上する。

地域化と正規表現特性

概要

文字列に対する正規表現マッチにおいて、ロカール依存の情報が必要になる場合がよくある。例えば、大文字小文字を区別しない比較はどのように行うのか？ ロカール依存の振る舞いは特性（traits）クラスが取り扱う。xpressive は cpp_regex_traits<>、c_regex_traits<>および null_regex_traits<>の 3 つの特性クラステンプレートを提供する。1 番目のものは std::locale をラップし、2 番目のものはグローバルな C ロカールをラップする。3 番目は非文字データを検索するのに使用する控えの特性型である。すべての特性テンプレートは正規表現特性のコンセプトに適合する。

既定の正規表現特性を設定する

既定では xpressive はすべてにパターンに対して cpp_regex_traits<>を使用する。これにより、すべての正規表現オブジェクトはグローバルな std::locale を使用する。BOOST_XPRESSIVE_USE_C_TRAITS を定義してコンパイルすると、xpressive の既定は c_regex_traits<>になる。

動的正規表現でカスタムの特性を使用する

カスタムの特性オブジェクトを使う動的正規表現を作成するには、regex_compiler<>を使わなければならない。基本的な方法を以下の例に示す。

9 この例を寄贈してくれた David Jenkins に感謝する。

```
// グローバルなCロカールを使うregex_compilerを宣言する
regex_compiler<char const *, c_regex_traits<char> > crxcomp;
cregex crx = crxcomp.compile( "\\w+" );

// カスタムのstd::localeを使うregex_compilerを宣言する
std::locale loc = /* ... ここでロカールを作成する ... */;
regex_compiler<char const *, cpp_regex_traits<char> > cpprxcomp(loc);
cregex cpprx = cpprxcomp.compile( "\\w+" );
```

regex_compilerオブジェクトは正規表現のファクトリとして動作する。これらは一度ロカールを与えておくと、以降作成する正規表現はそのロカールを使用するようになる。

静的正規表現でカスタムの特性を使用する

個々の静的正規表現に異なる特性群を使用したい場合は、imbue()特殊パターン修飾子を使用する。例えば、

```
// グローバルなCロカールを使う正規表現を定義する
c_regex_traits<char> ctraits;
sregex crx = imbue(ctraits)( +_w );

// カスタムのstd::localeを使う正規表現を定義する
std::locale loc = /* ... ここでロカールを作成する ... */;
cpp_regex_traits<char> cpptraits(loc);
sregex cpprx1 = imbue(cpptraits)( +_w );

// 上記の短縮形
sregex cpprx2 = imbue(loc)( +_w );
```

imbue()パターン修飾子はパターン全体を囲まなければならない。静的正規表現の一部だけを imbue するとエラーになる。例えば、

```
// エラー！ 正規表現の一部だけをimbue()することはできない
sregex error = _w >> imbue(loc)( _w );
```

null_regex_traits で非文字データを検索する

xpressive の静的正規表現では、パターンの検索は文字シーケンス内に限定されない。生のバイト、整数、その他文字のコンセプトに適合するものであれば何でも検索できる。このような場合、null_regex_traits<>を使うと簡単である。正規表現特性のコンセプトの控えの実装であり、文字クラスを無視し、大文字小文字に関する変換を一切行わない。

例えば整数列からパターンを検索する静的正規表現は、null_regex_traits<>を使って以下のように記述できる。

```
// 検索する整数データ
int const data[] = {0, 1, 2, 3, 4, 5, 6};

// 整数を検索するnull_regex_traits<>オブジェクトを作成する...
null_regex_traits<int> nul;
```

```
// 正規表現オブジェクトに null_regex_traits を指示する...
basic_regex<int const *> rex = imbue(nul)(1 >> +((set= 2,3) | 4) >> 5);
match_results<int const *> what;

// 整数の配列からパターンを検索する...
regex_search(data, data + 7, what, rex);

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

ヒント集

以下のヒント集に従うと、xpressive の効率を最大限に引き出せる。

パターンのコンパイルは一度とし、再利用せよ

正規表現のコンパイル（動的、静的によらない）は、マッチや検索の実行より **何倍もの**コストを要する。可能であれば basic_regex<>のコンパイルは一度だけにし、あとは再利用せよ（事あるごとに再作成してはならない）。

basic_regex<>オブジェクトはいかなる正規表現アルゴリズムによっても変更されないので、正規表現（と所属するすべての文法）の初期化が完了しさえすれば完全にスレッド安全である。パターンの再利用で一番簡単な方法は、basic_regex<>オブジェクトを static const にすることである。

match_results<>オブジェクトを再利用せよ

match_results<>オブジェクトは動的に確保したメモリをキャッシュする。そのため、正規表現検索を何度も行う場合は同じ match_results<>オブジェクトを再利用するほうがずっとよい。

注意：match_result<>オブジェクトはスレッド安全でないため、スレッドを超えて再利用してはならない。

match_results<>オブジェクトを引数に取るアルゴリズムを使用せよ

これも同様である。検索を複数回行う場合は、match_results<>オブジェクトを引数に取る正規表現アルゴリズムを使用し、毎回同じ match_results<>オブジェクトを使用すべきだ。match_results<>オブジェクトを与えないと一時オブジェクトが作成され、アルゴリズムが結果を返すときに破棄される。オブジェクトがキャッシュしていたメモリは解放され、次回また再確保されてしまう。

null 終端文字列に対してはイテレータの範囲を引数に取るアルゴリズムを使用せよ

xpressive は regex_match()および regex_search()アルゴリズムについて、C 形式の null 終端文字列を操作する多重定義を提供している。イテレータの範囲を引数に取る多重定義を使用すべきだ。null 終端文字列を正規表現アルゴリズムに渡すと、終端のイテレータを計算するために strlen が呼び出されてしまう。文字列の長さが事前に分かっているのであれば、[begin,end)組を取る正規表現を呼び出してこのオーバーヘッドを回避できる。

静的正規表現を使用せよ

静的正規表現は同じ内容の動的版に対して、平均で約 10%から 15%高速である。これだけでも静的版に慣れておく価値がある。

syntax_option_type::optimize を理解せよ

optimize フラグを正規表現コンパイラに渡すと、パターンの解析により多くの時間をかけるようになる。この結果、パターンによっては実行が高速になるが、コンパイル時間が長くなり、しばしばパターンが要するメモリの量が増える。パターンを再利用するのであれば optimize は効果があると考えてよい。パターンを一度しか使用しないのであれば、optimize は避けるべきだ。

よくある落とし穴

xpressive の落とし穴に足を踏み入れないように、以下のことを覚えておくとよい。

文法は単一のスレッドで作成せよ

静的正規表現では正規表現を入れ子にして文法を構築するが、外側の正規表現をコンパイルすると外側と内側の両方の正規表現オブジェクト、およびそれらが直接・間接的に参照するすべての正規表現オブジェクトが更新される。そのため、グローバルな正規表現オブジェクトが文法に関与すると危険である。単一のスレッドから正規表現文法を構築するのが最善である。一度構築してしまえば、正規表現文法は複数のスレッドから問題なく実行できる。

入れ子の数量子に注意せよ

これは多くの正規表現エンジンに共通の落とし穴であり、パターンによっては指数的に効率が悪化する。よくあるのは `(a*)*` のようにパターン内の数量子付きの項が他の数量子に入れ子になっているというものだが、多くの場合発見しにくいのが問題である。数量子が入れ子になっているパターンには注意せよ。

コンセプト

CharT の要件

型 BidiIterT を basic_regex<>のテンプレート引数とすると、iterator_traits<BidiIterT>::value_type が CharT である。型 CharT は自明な（trivial）既定コンストラクタ、コピーコンストラクタ、代入演算子、およびデストラクタをもたなければならない。さらにオブジェクトに関しては以下の要件を満たさなければならない。c は CharT 型、c1 と c2 は CharT const 型、i は int 型である。

表 14: CharT の要件

式	戻り値の型	表明、備考、事前・事後条件
CharT c	CharT	既定のコンストラクタ（自明でなければならない）。
CharT c(c1)	CharT	コピーコンストラクタ（自明でなければならない）。
c1 = c2	CharT	代入演算子（自明でなければならない）。
c1 == c2	bool	c1 の値が c2 と同じであれば true。
c1 != c2	bool	c1 と c2 が等値でなければ true。
c1 < c2	bool	c1 の値が c2 より小さければ true。
c1 > c2	bool	c1 の値が c2 より大きければ true。

c1 <= c2	bool	c1 が c2 より小さいか等値であれば true。
c1 >= c2	bool	c1 が c2 より大きいか等値であれば true。
intmax_t i = c1	int	CharT は整数型に変換可能でなければならない。
CharT c(i);	CharT	CharT は整数型から構築可能でなければならない。

特性の要件

以下の表において X は CharT 型の文字コンテナについて型と関数を定義する特性クラスである。u は X 型のオブジェクト、v は const X 型のオブジェクト、p は const CharT* 型の値、I1 と I2 は Input Iterator、c は const CharT 型の値、s は X::string_type 型のオブジェクト、cs は const X::string_type 型のオブジェクト、b は bool 型の値、i は int 型の値、F1 と F2 は const CharT*型の値、loc は X::locale_type 型のオブジェクト、ch は const char のオブジェクトである。

表 15: 特性の要件

式	戻り値の型	表明、備考、事前・事後条件
X::char_type	CharT	basic_regex<>クラステンプレートを実装する文字コンテナ型。
X::string_type	std::basic_string<CharT> か std::vector<CharT>	なし。
X::locale_type	(実装定義)	特性クラスが使用するロカールを表現する、コピー構築可能な型。
X::char_class_type	(実装定義)	個々の文字分類(文字クラス)を表現するビットマスク型。この型の複数の値をビット和すると別の有効な値を得る。
X::hash(c)	unsigned char	0 以上 UCHAR_MAX 以下の値を生成する。
v.widen(ch)	CharT	指定した char のワイド版を CharT で返す。
v.in_range(r1, r2, c)	bool	任意の文字 r1 と r2 について、r1 <= c && c <= r2 であれば true を返す。r1 <= r2 でなければならない。
v.in_range_nocase(r1, r2, c)	bool	任意の文字 r1 と r2 について、v.translate_nocase(d) == v.translate_nocase(c) かつ r1 <= d && d <= r2 となる文字 d が存在すれば true を返す。r1 <= r2 でなければならない。
v.translate(c)	X::char_type	c と等価、つまり v.translate(c) == v.translate(d) となるような文字 d を返す。
v.translate_nocase(c)	X::char_type	大文字小文字を区別せずに比較したとき c と等価、つまり v.translate_nocase(c) == v.translate_nocase(C) となるような文字 C。

v.transform(F1, F2)	X::string_type	イテレータ範囲[F1, F2)が示す文字シーケンスのソートキーを返す。文字シーケンス[G1, G2)が文字シーケンス[H1, H2)の前にソートされる場合にv.transform(G1, G2) < v.transform(H1, H2)とならなければならない。
v.transform_primary(F1, F2)	X::string_type	イテレータ範囲[F1, F2)が示す文字シーケンスのソートキーを返す。大文字小文字を区別せずにソートして文字シーケンス[G1, G2)が文字シーケンス[H1, H2)の前に現れる場合にv.transform_primary(G1, G2) < v.transform_primary(H1, H2)とならなければならない。
v.lookup_classname(F1, F2)	X::char_class_type	イテレータ範囲[F1, F2)が示す文字シーケンスを、isctypeに渡せるビットマスク型に変換する。lookup_classnameが返した値同士でビット和をとっても安全である。文字シーケンスがXが解釈できる文字クラス名でなければ0を返す。文字シーケンス内の大文字小文字の違いで戻り値が変化することはない。
v.lookup_collatename(F1, F2)	X::string_type	イテレータ範囲[F1, F2)が示す文字シーケンスが構成する照合要素を表す文字シーケンスを返す。文字シーケンスが正しい照合要素でなければ空文字列を返す。
v.isctype(c, v.lookup_classname(F1, F2))	bool	文字cが、イテレータ範囲[F1, F2)が示す文字クラスのメンバであれば真を返す。それ以外は偽を返す。
v.value(c, i)	int	文字cが基数iで有効な数字であれば、数字cの基数iでの数値を返す。[10]それ以外の場合は-1を返す。
u.imbue(loc)	X::locale_type	ロカールlocをuに指示する。uが直前まで使用していたロカールを返す。
v.getloc()	X::locale_type	vが使用中のロカールを返す。

謝辞

この節はBoost.Regexドキュメントの同じページと、正規表現を標準ライブラリに追加することになった草案をもとに作成した。

例

以下に6つの完全なプログラム例を挙げる。

10 iの値は8、10、16のいずれかである。

文字列全体が正規表現にマッチするか調べる

導入項にもあった例である。利便性のために再掲する。

```
#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'; // マッチ全体
        std::cout << what[1] << '\n'; // 1番目の捕捉
        std::cout << what[2] << '\n'; // 2番目の捕捉
    }

    return 0;
}
```

このプログラムは以下を出力する。

```
hello world!
hello
world
```

文字列が正規表現にマッチする部分文字列を含むか調べる

この例では、カスタムの mark_tag を使ってパターンを読みやすくしている点に注意していただきたい。後で mark_tag を match_results<>の添字に使っている。

```
#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.";

    // s1、s2、...よりも意味のある名前でカスタムの mark_tags を定義する
    mark_tag day(1), month(2), year(3), delim(4);

    // この正規表現は日付を検索する
    cregex date = (month= repeat<1,2>(_d))            // 先頭に月があり ...
                >> (delim= (set= '/','-'))            // その後ろに区切りがあり ...
```

```
                >> (day=   repeat<1,2>(_d)) >> delim  // さらに後ろに日と、同じ区切りがあり ...
                >> (year=  repeat<1,2>(_d >> _d));    // 最後に年がある。

    cmatch what;

    if( regex_search( str, what, date ) )
    {
        std::cout << what[0]     << '\n'; // マッチ全体
        std::cout << what[day]   << '\n'; // 日
        std::cout << what[month] << '\n'; // 月
        std::cout << what[year]  << '\n'; // 年
        std::cout << what[delim] << '\n'; // 区切り
    }

    return 0;
}
```

このプログラムは以下を出力する。

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

正規表現にマッチした部分文字列をすべて置換する

以下のプログラムは文字列内の日付を検索し、擬似 HTML でマークアップする。

```
#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." );

    // 本質的には前の例と同じ正規表現だが、動的正規表現を使っている
    sregex date = sregex::compile( "(\\d{1,2})([/-])(\\d{1,2})\\2((?:\\d{2}){1,2})" );

    // Perl と同様、$&は正規表現にマッチした部分文字列を参照する
    std::string format( "<date>$&</date>" );

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

    return 0;
}
```

このプログラムは以下を出力する。

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

正規表現にマッチする部分文字列をすべて検索し、1 つずつ辿る

以下のプログラムはワイド文字列から単語を検索する。wsregex_iterator を使う。wsregex_iterator を参照はがしするとwsmatch オブジェクトが得られることに注意していただきたい。

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

using namespace boost::xpressive;

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

    // 単語全体を検索する
    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;
}
```

このプログラムは以下を出力する。

```
This
is
his
face
```

文字列をそれぞれ正規表現にマッチするトークンに分割する

以下のプログラムは文字列からレースのタイムを検索し、はじめに分、次に秒を表示する。regex_token_iterator<>を使っている。

```
#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" );
```

```
    // レースのタイムを検索する
    sregex time = sregex::compile( "(\\d):(\\d\\d)" );

    // 各マッチについて、トークンイテレータは始めに 1 番目のマーク済み部分式の値
    // 次に 2 番目のマーク済み部分式の値をとらなければならない
    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;
}
```

このプログラムは以下を出力する。

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

正規表現を区切りとして文字列を分割する

以下のプログラムは HTML でマークアップされたテキストからマークアップを除去する。HTML タグにマッチする正規表現と、文字列内の正規表現にマッチ**しなかった**部分を返す `regex_token_iterator<>`を使っている。

```
#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." );

    // HTML タグを検索する
    sregex html = '<' >> optional('/') >> +_w >> '>';

    // 以下のようにトークンイテレータに-1 を与えると
    // 正規表現にマッチ*しなかった*文字列部分を表示する。
    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;
}
```

このプログラムは以下を出力する。

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

入れ子になった結果木を表示する

入れ子になった結果木を表示する方法を以下のヘルパクラスで示す。

```
// 入れ子になった結果を字下げ付きで std::cout に出力する
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
    {
        // はじめに字下げする
        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 );

        // マッチを出力する
        std::cout << what[0] << '\n';

        // 入れ子のマッチを出力する
        std::for_each(
            what.nested_results().begin(),
            what.nested_results().end(),
            output_nested_results( tabs_ + 1 ) );
    }
};
```

リファレンス

<boost/xpressive/basic_regex.hpp>ヘッダ

basic_regex<>クラステンプレートの定義と、関連するヘルパ関数がある。

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

basic_regex 構造体テンプレート

boost::xpressive::basic_regex – basic_regex<>クラステンプレートはコンパイル済み正規表現を保持するクラスである。

書式

```
// ヘッダ：<boost/xpressive/basic_regex.hpp>

template<typename BidiIter>
struct basic_regex {
  // 型
  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;

  // 構築、コピー、解体
  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 &);

  // 公開メンバ関数
  std::size_t mark_count() const;
  regex_id_type regex_id() const;
  void swap(basic_regex< BidiIter > &);

  // 公開静的メンバ関数
  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);
  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;
};
```

説明

basic_regex 構築、コピー、解体の公開演算

```
basic_regex();
```

事後条件: `regex_id() == 0`

`mark_count() == 0`

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

引数: *that* コピーする `basic_regex` オブジェクト。

事後条件: `regex_id() == that.regex_id()`

`mark_count() == that.mark_count()`

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

静的正規表現から構築する。

引数: *expr* 静的正規表現。

要件: `Expr` は静的正規表現の型。

事後条件: `regex_id() != 0`

`mark_count() >= 0`

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

引数: *that* コピーする `basic_regex` オブジェクト。

事後条件: `regex_id() == that.regex_id()`

mark_count() == that.mark_count()

戻り値: *this

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

静的正規表現から構築する。

引数: *expr* 静的正規表現。

要件: Expr は静的正規表現の型。

事後条件: regex_id() != 0

mark_count() >= 0

戻り値: *this

例外: std::bad_alloc

basic_regex の公開メンバ関数

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

この正規表現内の捕捉済み部分式の数を返す。

```
regex_id_type regex_id() const;
```

この正規表現を一意に識別するトークンを返す。

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

この basic_regex オブジェクトの内容を別のものと交換する。

引数: *that* 他の basic_regex オブジェクト。

例外: 例外を送出しない。

備考: 参照まで追跡しない浅い交換である。basic_regex オブジェクトを参照により別の正規表現に組み込み、他の basic_regex オブジェクトと内容を交換すると、外側の正規表現からはこの変更を検出できない。これは swap() が例外を送出できないためである。

basic_regex の公開静的メンバ関数

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

文字の範囲から正規表現オブジェクトを構築するファクトリメソッド。regex_compiler< BidiIter >().compile(begin,

end, flags);と等価。

引数: *begin* コンパイルする正規表現を表す文字範囲の先頭。
end コンパイルする正規表現を表す文字範囲の終端。
flags 文字列をどのように解釈するかを指定する省略可能なビットマスク（syntax_option_typeを見よ）。

要件: [begin, end)が有効な範囲である。
[begin, end)で指定した文字の範囲が正規表現の有効な文字列表現である。

戻り値: 文字の範囲が表す正規表現に相当するbasic_regexオブジェクト。

例外: regex_error

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

利便性のためのメンバ関数多重定義。上記関数と受け取る引数が異なるのみ。

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

利便性のためのメンバ関数多重定義。上記関数と受け取る引数が異なるのみ。

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

利便性のためのメンバ関数多重定義。上記関数と受け取る引数が異なるのみ。

swap 関数テンプレート

boost::xpressive::swap – 2つのbasic_regexオブジェクトの内容を交換する。

書式

```
// ヘッダ：<boost/xpressive/basic_regex.hpp>


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

説明

引数: *left* 第1のbasic_regexオブジェクト。
right 第2のbasic_regexオブジェクト。

例外： 送出しない。

備考： 参照まで追跡しない浅い交換である。basic_regex オブジェクトを参照により別の正規表現に組み込み、他の basic_regex オブジェクトと内容を交換すると、外側の正規表現からはこの変更を検出できない。これは swap() が例外を送出できないためである。

<boost/xpressive/match_results.hpp>ヘッダ

match_results 型の定義と、関連するヘルパがある。match_results 型は regex_match() および regex_search() 操作の結果を保持する。

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

match_results 構造体テンプレート

Boost::xpressive::match_results – match_results<>クラステンプレートは regex_match() や regex_search() の結果を sub_match オブジェクトのコレクションとして保持する。

書式

```
// ヘッダ：<boost/xpressive/match_results.hpp>

template<typename BidiIter>
struct match_results {
  // 型
  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;

  // 構築、コピー、解体
  match_results();
  match_results(match_results< BidiIter > const &);
  match_results< BidiIter >& operator=(match_results< BidiIter > const &);
  ~match_results();

  // 公開メンバ関数
  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 &);
};
```

説明

クラステンプレート match_results<>は、正規表現マッチの結果を表すシーケンスのコレクションである。コレクションの領域は match_results<>クラスのメンバ関数が必要に応じて確保・解放する。

クラステンプレート match_results<>は、lib.sequence.reqmts が規定するシーケンスの要件に適合するが、const なシーケンスに対して定義された演算だけをサポートする。

match_results の構築、コピー、解体公開演算

```
match_results();
```

事後条件： regex_id() == 0
size() == 0
empty() == true
str() == string_type()

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

引数: *that* コピーする match_results オブジェクト。

事後条件: regex_id() == that.regex_id()

size() == that.size()

empty() == that.empty()

n < that.size() であるすべての自然数 *n* について str(n) == that.str(n)

prefix() == that.prefix()

suffix() == that.suffix()

n < that.size() であるすべての自然数 *n* について (*this)[n] == that[n]

n < that.size() であるすべての自然数 *n* について length(n) == that.length(n)

n < that.size() であるすべての自然数 *n* について position(n) == that.position(n)

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

引数: *that* コピーする match_results オブジェクト。

事後条件: regex_id() == that.regex_id()

size() == that.size()

empty() == that.empty()

n < that.size() であるすべての自然数 *n* について str(n) == that.str(n)

prefix() == that.prefix()

suffix() == that.suffix()

n < that.size() であるすべての自然数 *n* について (*this)[n] == that[n]

n < that.size() であるすべての自然数 *n* について length(n) == that.length(n)

n < that.size() であるすべての自然数 *n* について position(n) == that.position(n)

```
~match_results();
```

match_results 公開メンバ関数

```
size_type size() const;
```

*this が成功したマッチ結果を表す場合は、マッチしたマーク済み部分式の総数に 1 を足した数を返す。それ以外の場合は 0 を返す。

```
bool empty() const;
```

size() == 0 を返す。

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

(*this)[sub].length()を返す。

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

!(*this)[sub].matched であれば-1 を返す。それ以外の場合は std::distance(base, (*this)[sub].first)を返す（base は検索対象のシーケンスの開始イテレータ）。[11]

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

(*this)[sub].str()を返す。

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

マーク済み部分式 *sub* にマッチしたシーケンスを表す sub_match オブジェクトへの参照を返す。sub == 0 であれば正規表現全体にマッチしたシーケンスを表す sub_match オブジェクトへの参照を返す。sub >= size()であればマッチしなかった部分式を表す sub_match オブジェクトへの参照を返す。

```
const_reference prefix() const;
```

マッチ・検索対象文字列の先頭からマッチが見つかった位置までの文字シーケンスを表す sub_match オブジェクトへの参照を返す。

要件： (*this)[0].matched が真

```
const_reference suffix() const;
```

マッチが見つかった位置の終端からマッチ・検索対象文字列の終端までの文字シーケンスを表す sub_match オブジェクトへの参照を返す。

要件： (*this)[0].matched が真

```
const_iterator begin() const;
```

*this に格納されたマーク済み部分式マッチをすべて列挙する開始イテレータを返す。

```
const_iterator end() const;
```

*this に格納されたマーク済み部分式マッチをすべて列挙する終了イテレータを返す。

11 regex_iterator による繰り返し検索の途中でなければ、base は prefix().first と同じである。

```
operator bool_type() const;
```

(*this)[0].matched であれば真を、そうでなければ偽を返す。

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

empty() || !(*this)[0].matched であれば真を、そうでなければ偽を返す。

```
regex_id_type regex_id() const;
```

この match_results オブジェクトで最近使用した basic_regex オブジェクトの識別子を返す。

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

入れ子の match_results 要素のシーケンスを返す。

```
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;
```

Format が ForwardRange か null 終端文字列であれば、*fmt* 内の文字シーケンスを OutputIterator である *out* にコピーする。*fmt* 内の各書式化子およびエスケープシーケンスについて、それらが表す文字（列）かそれらが参照する *this 内のシーケンスで置換する。*flags* で指定したビットマスクは、どの書式化子あるいはエスケープシーケンスを使用するかを決定する。既定では『ECMA-262、ECMAScript 言語仕様 15 章 5.4.11 String.prototype.replace』が使用する書式である。

それ以外で *Format* が Callable<match_results<BidiIter>, OutputIterator, regex_constants::match_flag_type>であれば、この関数は fmt(*this, out, flags)を返す。

それ以外で *Format* が Callable<match_results<BidiIter>, OutputIterator>であれば、この関数は fmt(*this, out)を返す。

それ以外で *Format* が Callable<match_results<BidiIter> >であれば、この関数は std::copy(x.begin(), x.end(), out)を返す。*x* は fmt(*this)を呼び出した結果である。

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

利便性のために提供している多重定義メンバ関数である。上記関数とは受け取る引数が異なるのみである。

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

```
                    unspecified = 0) const;
```

Format が ForwardRange か null 終端文字列であれば、この関数は文字シーケンス *fmt* のコピーを返す。*fmt* 内の各書式化子およびエスケープシーケンスについて、それらが表す文字（列）かそれらが参照する`*this` 内のシーケンスで置換する。*flags* で指定したビットマスクは、どの書式化子あるいはエスケープシーケンスを使用するかを決定する。既定では『ECMA-262、ECMAScript 言語仕様 15 章 5.4.11 String.prototype.replace』が使用する書式である。

それ以外で *Format* が `Callable<match_results<BidiIter>, OutputIterator, regex_constants::match_flag_type>`であれば、この関数は `fmt(*this, out, flags)` 呼び出しで得られた `string_type` オブジェクト `x` を返す。*out* は `x` への `back_insert_iterator` である。

それ以外で *Format* が `Callable<match_results<BidiIter>, OutputIterator>`であれば、この関数は `fmt(*this, out)` の呼び出しで得られた `string_type` オブジェクト `x` を返す。out は `x` への `back_insert_iterator` である。

それ以外で *Format* が `Callable<match_results<BidiIter> >`であれば、この関数は `fmt(*this)` を返す。

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

利便性のために提供している多重定義メンバ関数である。上記関数とは受け取る引数が異なるのみである。

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

2 つの `match_results` オブジェクトの内容を交換する。例外を投げないことを保証する。

引数: *that* 交換する `match_results` オブジェクト。

事後条件: `*this` が *that* 内にあった部分式マッチのシーケンスをもつ。*that* が`*this` 内にあった部分式マッチのシーケンスをもつ。

例外: 送出しない。

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

TODO document me[12]

regex_id_filter_predicate 構造体テンプレート

boost::xpressive::regex_id_filter_predicate。

12 訳注 この節はまだ原文がありません。

書式

```
// ヘッダ：<boost/xpressive/match_results.hpp>

template<typename BidiIter>
struct regex_id_filter_predicate {
  // 構築、コピー、解体
  regex_id_filter_predicate(regex_id_type);

  // 公開メンバ関数
  bool operator()(match_results< BidiIter > const &) const;
};
```

説明

regex_id_filter_predicate 構築、コピー、解体の公開演算

```
regex_id_filter_predicate(regex_id_type regex_id);
```

regex_id_filter_predicate 公開メンバ関数

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

<boost/xpressive/regex_actions.hpp>ヘッダ

xpressive におけるアクション式の構文要素がある。

```
BOOST_PROTO_LOCAL_MACRO(N, typename_A, A_const_ref, A_const_ref_a, a)
BOOST_PROTO_LOCAL_a
BOOST_PROTO_LOCAL_LIMITS
```

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

    function< op::at >::type const at;
    function< op::push >::type const push;
    function< op::push_back >::type const push_back;
    function< op::push_front >::type const push_front;
    function< op::pop >::type const pop;
    function< op::pop_back >::type const pop_back;
```

```
    function< op::pop_front >::type const pop_front;
    function< op::top >::type const top;
    function< op::back >::type const back;
    function< op::front >::type const front;
    function< op::first >::type const first;
    function< op::second >::type const second;
    function< op::matched >::type const matched;
    function< op::length >::type const length;
    function< op::str >::type const str;
    function< op::insert >::type const insert;
    function< op::make_pair >::type const make_pair;
    unspecified check;
    unspecified let;
    template<typename X2_0, typename A0> unspecified as(A0 const &);
    template<typename X2_0, typename A0> unspecified static_cast_(A0 const &);
    template<typename X2_0, typename A0> unspecified dynamic_cast_(A0 const &);
    template<typename X2_0, typename A0> unspecified const_cast_(A0 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 &);
    namespace op {
      struct push;
      struct push_back;
      struct push_front;
      struct pop;
      struct pop_back;
      struct pop_front;
      struct front;
      struct back;
      struct top;
      struct first;
      struct second;
      struct matched;
      struct length;
      struct str;
      struct insert;
      struct make_pair;
      template<typename T> struct as;
      template<typename T> struct static_cast_;
      template<typename T> struct dynamic_cast_;
      template<typename T> struct const_cast_;
      template<typename T> struct construct;
      template<typename Except> struct throw_;
    }
  }
}
```

（訳注）以下の実体を省略しました。

- マクロ：BOOST_PROTO_LOCAL_MACRO、BOOST_PROTO_LOCAL_a、BOOST_PROTO_LOCAL_LIMITS
- 構造体テンプレート：function、value、local、as、static_cast_、dynamic_cast_、const_cast_、construct、throw_
- グローバル定数：at、push、push_back、push_front、pop、pop_back、pop_front、top、back、front、first、second、matched、length、str、insert、make_pair
- グローバル変数：check、let
- 関数テンプレート：as（`lexical_cast` のようなもの）、static_cast_、dynamic_cast_、const_cast_、val、ref、cref

- 構造体：push、push_back、push_front、pop、pop_back、pop_front、front、back、top、first、second、matched、length、str、insert、make_pair

placeholder 構造体テンプレート

boost::xpressive::placeholder。

書式

```
// ヘッダ：<boost/xpressive/regex_actions.hpp>

template<typename T, int I, typename Dummy>
struct placeholder {
  // 型
  typedef placeholder< T, I, Dummy > this_type;
  typedef unspecified                action_arg_type;
};
```

説明

placeholder<T>は、意味アクションにおいて型 T の変数情報を表すプレースホルダを定義するのに使用する。

<boost/xpressive/regex_algorithms.hpp>ヘッダ

regex_match()、regex_search()および regex_replace()アルゴリズムがある。

```
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);
  }
}
```

regex_match 関数

boost::xpressive::regex_match – 正規表現がシーケンスの先頭から終端までにマッチするか調べる。

書式

```
// ヘッダ：<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);
```

説明

正規表現 `re` とシーケンス `[begin, end)` 全体の間に完全なマッチがあるか確定する。

引数:

begin	シーケンスの先頭。
end	シーケンスの終端。
flags	正規表現をシーケンスに対してどのようにマッチさせるか制御する、省略可能なマッチフラグ（`match_flag_type` を見よ）。
re	使用する正規表現オブジェクト。
rng	シーケンス。
what	`sub_match` を書き込む `match_results` 構造体。

要件: 型 BidiIter が双方向イテレータ（24.1.4）の要件を満たす。

`[begin, end)` が有効なイテレータ範囲を表す。

戻り値: マッチが見つかった場合は `true`、それ以外の場合は `false`。

例外: `regex_error`

regex_search 関数

boost::xpressive::regex_search – `[begin, end)` 内に、正規表現 `re` にマッチする部分シーケンスがあるか調べる。

書式

```
// ヘッダ：<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);
```

説明

[begin, end) 内に、正規表現 re にマッチする部分シーケンスがあるか確定する。

引数:

begin	シーケンスの先頭。
end	シーケンスの終端。
rng	シーケンス。
what	sub_match を書き込む match_results 構造体。
re	使用する正規表現オブジェクト。
flags	正規表現をシーケンスに対してどのようにマッチさせるか制御する、省略可能なマッチフラグ（match_flag_type を見よ）。

要件: 型 *BidiIter* が双方向イテレータ（24.1.4）の要件を満たす。

[begin, end) が有効なイテレータ範囲を表す。

戻り値: マッチが見つかった場合は true、それ以外の場合は false。

例外: regex_error

regex_replace 関数

boost::xpressive::replace – 与えられた入力シーケンス、正規表現、および書式化文字列、書式化オブジェクト、関数、式に対して

出力シーケンスを構築する。

書式

```
// ヘッダ：<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);
```

説明

`regex_iterator< BidiIter > i(begin, end, re, flags)` で regex_iterator オブジェクトを構築し、シーケンス

[begin, end)に現れるmatch_results< BidiIter >型のマッチmすべてをiを使って列挙する。マッチが見つからず、かつ!(flags & format_no_copy)であればstd::copy(begin, end, out)を呼び出す。それ以外の場合は、見つかった各マッチについて!(flags & format_no_copy)であればstd::copy(m.prefix().first, m.prefix().second, out)を呼び出し、次にm.format(out, format, flags)を呼び出す。最後に!(flags & format_no_copy)であればstd::copy(last_m.suffix().first, last_m.suffix().second, out)を呼び出す（last_mは最後に見つかったマッチのコピー）。

flags & format_first_onlyが非ゼロの場合は、最初に見つかったマッチのみを置換する。

引数:

begin	入力シーケンスの先頭。
end	入力シーケンスの終端。
flags	正規表現をシーケンスに対してどのようにマッチさせるか制御する、省略可能なマッチフラグ（match_flag_typeを見よ）。
format	置換シーケンスを整形する書式化文字列。または書式化関数、オブジェクト、式。
out	出力シーケンスを書き込む出力イテレータ。
re	使用する正規表現オブジェクト。
str	入力シーケンス。

要件: BidiIterが双方向イテレータ(24.1.4)の要件を満たす。

OutIterが出力イテレータ(24.1.2)の要件を満たす。

Formatter型がForwardRange、Callable<match_results<BidiIter> >, Callable<match_results<BidiIter>, OutIter>あるいはCallable<match_results<BidiIter>, OutIter, regex_constants::match_flag_type>のいずれか。またはnull終端書式化文字列か書式化ラムダ式を表す式テンプレート。

[begin, end)が有効なイテレータ範囲を表す。

戻り値: 出力シーケンスを書き込んだ後の出力イテレータ。

例外: regex_error

<boost/xpressive/regex_compiler.hpp>ヘッダ

正規表現を文字列から構築するファクトリであるregex_compilerの定義がある。

```
namespace boost {
  namespace xpressive {
    template<typename BidiIter, typename RegexTraits, typename CompilerTraits>
      struct regex_compiler;
  }
}
```

regex_compiler 構造体テンプレート

boost::xpressive::regex_compiler – regex_compilerクラステンプレートは文字列からbasic_regexオブジェクトを構築するファ

クトリである。

書式

```
// ヘッダ：<boost/xpressive/regex_compiler.hpp>

template<typename BidiIter, typename RegexTraits, typename CompilerTraits>
struct regex_compiler {
  // 型
  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;

  // 構築、コピー、解体
  regex_compiler(RegexTraits const & = RegexTraits());

  // 公開メンバ関数
  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;
};
```

説明

regex_compiler クラステンプレートは、文字列から basic_regex オブジェクトを構築するのに使用する。文字列は正しい正規表現でなければならない。regex_compiler オブジェクトにロカールを指示すると、以降その regex_compiler オブジェクトが作成する basic_regex オブジェクトはすべてそのロカールを使用する。regex_compiler オブジェクト作成後、（必要であればロカールを与え、）正規表現を表す文字列を使って compile() メソッドを呼び出すことで basic_regex オブジェクトを構築する。同じ regex_compiler オブジェクトに対して compile() は複数回呼び出すことができる。同じ文字列からコンパイルした 2 つの basic_regex オブジェクトは異なる regex_id をもつ。

regex_compiler 構築、コピー、解体の公開演算

```
regex_compiler(RegexTraits const & traits = RegexTraits());
```

regex_compiler 公開メンバ関数

```
locale_type imbue(locale_type loc);
```

regex_compiler が使用するロカールを指定する。

引数: *loc* この regex_compiler が使用するロカール。

戻り値: 直前のロカール。

```
locale_type getloc() const;
```

regex_compiler が使用しているロカールを返す。

戻り値: この regex_compiler が使用しているロカール。

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

文字の範囲から basic_regex オブジェクトを構築する。

引数: *begin* コンパイルする正規表現を表す文字範囲の先頭。

end コンパイルする正規表現を表す文字範囲の終端。

flags パターン文字列をどのように解釈するか決める省略可能なビットマスク（syntax_option_type を見よ）。

要件: InputIter が入力イテレータの要件を満たす。

[begin, end) が有効な範囲である。

[begin, end) で指定した文字範囲が正しい正規表現の文字列表現である。

戻り値: 文字範囲が表す正規表現に相当する basic_regex オブジェクト。

例外: regex_error

```
template<typename InputRange>
  disable_if< is_pointer< InputRange >, basic_regex< BidiIter > >::type
  compile(InputRange const & pat,
          flag_type flags = regex_constants::ECMAScript);
```

利便性のためのメンバ関数多重定義。上記関数と受け取る引数が異なるのみ。

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

利便性のためのメンバ関数多重定義。上記関数と受け取る引数が異なるのみ。

```
basic_regex< BidiIter >
compile(char_type const * begin, std::size_t size, flag_type flags);
```

利便性のためのメンバ関数多重定義。上記関数と受け取る引数が異なるのみ。

```
basic_regex< BidiIter > & operator[](string_type const & name);
```

名前付き正規表現への参照を返す。指定した名前をもつ正規表現が存在しない場合は、新しい正規表現を作成し参照を返す。

引数: *name*　正規表現の名前を表す `std::string`。

要件: 文字列が空でない。

例外: `bad_alloc`

```
basic_regex< BidiIter > const & operator[](string_type const & name) const;
```

利便性のためのメンバ関数多重定義。上記関数と受け取る引数が異なるのみ。

<boost/xpressive/regex_constants.hpp>ヘッダ

`syntax_option_type`、`match_flag_type` および `error_type` 列挙の定義がある。

```
namespace boost {
  namespace xpressive {
    namespace regex_constants {
      enum syntax_option_type;
      enum match_flag_type;
      enum error_type;
    }
  }
}
```

syntax_option_type 型

boost::xpressive::regex_constants::syntax_option_type。

書式

```
// ヘッダ：<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 };
```

説明

正規表現構文をカスタマイズするのに使用するフラグ。

ECMAScript	正規表現エンジンが通常の文法を使用するよう指定する。この文法は ECMA-262、ECMAScript 言語仕様 15 章 10 RegExp (Regular Expression) Objects（FWD.1）に示されているものと同じである。
icase	文字コンテナシーケンスに対して正規表現マッチを大文字小文字を区別せずに行うことを指定する。
nosubs	正規表現が文字コンテナシーケンスに対してマッチしたとき、与えられた match_results 構造体に部分式マッチを格納しないことを指定する。
optimize	正規表現エンジンに対して、マッチの高速化により注意を払うよう指定する。これを行うと正規表現オブジェクトの構築速度が低下する。検出不可能な作用がプログラム出力に現れることはない。
collate	[a-b]形式の文字範囲がロカールを考慮することを指定する。
single_line	メタ文字^および$が内部の改行にマッチしないことを指定する。これは Perl の既定と逆であることに注意していただきたい。Perl の/m（複数行）修飾子と反対である。
not_dot_null	メタ文字.が null 文字\0にマッチしないことを指定する。
not_dot_newline	メタ文字.が改行文字\nにマッチしないことを指定する。
ignore_white_space	エスケープされていない空白類を無視するよう指定する。

match_flag_type 型

boost::xpressive::regex_constants::match_flag_type。

書式

```
// ヘッダ：<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 };
```

説明

正規表現アルゴリズムの動作をカスタマイズするのに使用するフラグ。

match_default	ECMA-262、ECMAScript 言語仕様 15 章 10 RegExp (Regular Expression) Objects（FWD.1）の通常規則に一切の変更を加えることなく正規表現マッチを行うことを指定する。
match_not_bol	正規表現^が部分シーケンス[first,first)にマッチしないことを指定する。

match_not_eol	正規表現$が部分シーケンス[last,last)にマッチしないことを指定する。
match_not_bow	正規表現\bが部分シーケンス[first,first)にマッチしないことを指定する。
match_not_eow	正規表現\bが部分シーケンス[last,last)にマッチしないことを指定する。
match_any	複数のマッチが可能な場合、それらのいずれでも結果として扱ってよいことを指定する。
match_not_null	正規表現が空のシーケンスにマッチしないことを指定する。
match_continuous	正規表現が first を先頭とする部分シーケンスにマッチしなければならないことを指定する。
match_partial	マッチが見つからない場合、from != last であるマッチ[from, last)を結果として返すことを指定する([from, last)を接頭辞とするより長い文字シーケンス[from, to)が完全マッチの結果として存在する可能性がある場合)。
match_prev_avail	--first が有効なイテレータ位置であることを指定する。このフラグを設定すると、正規表現アルゴリズム(RE.7)およびイテレータ(RE.8)は match_not_bol および match_not_bow フラグを無視する。[13]
format_default	正規表現マッチを新しい文字列で置換するとき、ECMA-262、ECMAScript 言語仕様 15 章 5.4.11 String.prototype.replace(FWD.1)が規定する ECMAScript の replace 関数が使用する規則を用いて新しい文字列を構築する。検索置換操作については、互いに重複しない正規表現マッチを検索・置換し、正規表現にマッチしなかった入力部分を変更することなく出力文字列にコピーする。
format_sed	正規表現マッチを新しい文字列で置換するとき、 IEEE Std 1003.1-2001, Portable Operating SystemInterface (POSIX), Shells and Utilities が規定する Unix sed ユーティリティが使用する規則を用いて新しい文字列を構築する。
format_perl	正規表現マッチを新しい文字列で置換するとき、ECMA-262、ECMAScript 言語仕様 15 章 5.4.11 String.prototype.replace(FWD.1)が規定する ECMAScript の replace 関数の規則のスーパーセットを使って新しい文字列を構築する。
format_no_copy	検索置換操作で指定すると、マッチを行う文字コンテナシーケンスを出力文字列にコピーしない。
format_first_only	検索置換操作で指定すると、最初の正規表現マッチのみを置換する。
format_literal	書式化文字列をリテラルとして扱う。
format_all	条件置換(?ddexpression1:expression2)を含むすべての構文拡張を有効にする。

error_type 型

boost::xpressive::regex_constants::error_type。

13 訳注 “RE.n” は N1429 の節番号(http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1429.htm)。

書式

```
// ヘッダ：<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 };
```

説明

regex_error が使用するエラーコード。

error_collate	正規表現内に不正な照合要素名がある。
error_ctype	正規表現内に不正な文字クラス名がある。
error_escape	正規表現内に不正なエスケープ付き文字があるか単独のエスケープがある。
error_subreg	正規表現内に不正な後方参照がある。
error_brack	正規表現内に一致しない[と]がある。
error_paren	正規表現内に一致しない(と)がある。
error_brace	正規表現内に一致しない{と}がある。
error_badbrace	正規表現内の{}式に不正な範囲がある。
error_range	正規表現内に不正な文字範囲がある（例：[b-a]）。
error_space	正規表現を有限状態マシンに変換するのにメモリが不足している。
error_badrepeat	*、?、+、{のいずれかが正しい正規表現の後にない。
error_complexity	マッチを行う正規表現の計算量が規定の水準を超えた。
error_stack	指定した文字シーケンスに対して正規表現がマッチするか、を決定するのに必要なメモリが不足している。
error_badref	入れ子の正規表現が未初期化である。
error_badmark	名前付き捕捉の使用が不正。
error_badlookbehind	可変幅の後方先読み表明の作成を検出した。
error_badrule	規則の不正な使用を検出した。
error_badarg	アクションの引数が束縛されていない。
error_badattr	未初期化の属性を読み取ろうとした。
error_internal	内部エラーが発生した。

<boost/xpressive/regex_error.hpp>ヘッダ

regex_error 例外クラスの定義がある。

```
BOOST_XPR_ENSURE_(pred, code, msg)
```

```
namespace boost {
  namespace xpressive {
    struct regex_error;
  }
}
```

（訳注）以下の実体を省略しました。

- マクロ：BOOST_XPR_ENSURE_

regex_error 構造体

boost::xpressive::regex_error – regex_error クラスは、正規表現を表す文字列を有限状態マシンに変換するときに、エラーを報告するのに送出する例外オブジェクトの型を定義する。

書式

```
// ヘッダ：<boost/xpressive/regex_error.hpp>


struct regex_error : public std::runtime_error {
  // 構築、コピー、解体
  regex_error(regex_constants::error_type, char const * = "");
  ~regex_error();

  // 公開メンバ関数
  regex_constants::error_type code() const;
};
```

説明

regex_error 構築、コピー、解体の公開演算

```
regex_error(regex_constants::error_type code, char const * str = "");
```

regex_error クラスのオブジェクトを構築する。

引数: *code* regex_error が表す error_type。

事後条件： `code() == code`

```
~regex_error();
```

`regex_error` クラスのデストラクタ。

例外： 例外は送出しない。

regex_error 公開メンバ関数

```
regex_constants::error_type code() const;
```

`error_type` 値のアクセス子。

戻り値： コンストラクタに渡した `error_type` コード。

例外： 例外は送出しない。

<boost/xpressive/regex_iterator.hpp>ヘッダ

シーケンス内のすべてのマッチを辿る STL 互換のイテレータである `regex_iterator` 型の定義がある。

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

regex_iterator 構造体テンプレート

boost::xpressive::regex_iterator。

書式

```
// ヘッダ：<boost/xpressive/regex_iterator.hpp>

template<typename BidiIter>
struct regex_iterator {
  // 型
  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;

  // 構築、コピー、解体
```

```
  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 &);

  // 公開メンバ関数
  value_type const & operator*() const;
  value_type const * operator->() const;
  regex_iterator< BidiIter > & operator++();
  regex_iterator< BidiIter > operator++(int);
```

説明

regex_iterator 構築、コピー、解体の公開演算

（訳注）5 つの実体を省略します。

regex_iterator 公開メンバ関数

（訳注）3 つの実体を省略します。

```
regex_iterator< BidiIter > & operator++();
```

what.prefix().first != what[0].second かつ match_prev_avail 要素がフラグに設定されていなければ設定する。次に regex_search(what[0].second, end, what, *pre, flags) を呼び出したのと同様の振る舞いをする。ただし以下の点については振る舞いが異なる。前回見つかったマッチがゼロ幅（what[0].length() == 0）の場合、what[0].second を先頭とする非ゼロ幅マッチを探索する。これが失敗し、かつ what[0].second != suffix().second である場合に限り what[0].second + 1 を先頭とするマッチ（これもゼロ幅かもしれない）を探索する。それ以上マッチがない場合は this をシーケンスの終端イテレータと等値に設定する。

事後条件: (*this)->size() == pre->mark_count() + 1

(*this)->empty() == false

(*this)->prefix().first ==（前回見つかったマッチの終端位置を指すイテレータ）

(*this)->prefix().last == (**this)[0].first

(*this)->prefix().matched == (*this)->prefix().first != (*this)->prefix().second

(*this)->suffix().first == (**this)[0].second

(*this)->suffix().last == end

(*this)->suffix().matched == (*this)->suffix().first != (*this)->suffix().second

(**this)[0].first ==（このマッチの文字列イテレータ）

`(**this)[0].second` == （このマッチの終端イテレータ）

`(**this)[0].matched` == （完全マッチが見つかった場合は真、（`match_partial` フラグを設定して）部分マッチの場合は偽）

`(**this)[n].first` == （`n < (*this)->size()` であるあらゆる整数について、*n* 番目の部分式にマッチしたシーケンスの先頭。*n* 番目の部分式がマッチに関与しなかった場合は `end`）

`(**this)[n].second` == （`n < (*this)->size()` であるあらゆる整数について、*n* 番目の部分式にマッチしたシーケンスの終端。*n* 番目の部分式がマッチに関与しなかった場合は `end`）

`(**this)[n].matched` == （`n < (*this)->size()` であるあらゆる整数について、*n* 番目の部分式がマッチに関与していれば真、それ以外の場合は偽）

`(*this)->position()` == （走査対象シーケンスの先頭からこのマッチの先頭までの距離）

<boost/xpressive/regex_primitives.hpp>ヘッダ

静的正規表現を記述するための構文要素がある。

```
namespace boost {
  namespace xpressive {
    struct mark_tag;

    unsigned int const inf;     // 部分式の無限回の繰り返しに使う。
    unspecified nil;     // 空のマッチ。
    unspecified alnum;     // 英数字にマッチ。
    unspecified alpha;     // アルファベットにマッチ。
    unspecified blank;     // 空白（水平空白）文字にマッチ。
    unspecified cntrl;     // 制御文字にマッチ。
    unspecified digit;     // 数字にマッチ。
    unspecified graph;     // グラフィカルな文字にマッチ。
    unspecified lower;     // 小文字にマッチ。
    unspecified print;     // 印字可能な文字にマッチ。
    unspecified punct;     // 区切り文字にマッチ。
    unspecified space;     // 空白類文字にマッチ。
    unspecified upper;     // 大文字にマッチ。
    unspecified xdigit;     // 16進数字にマッチ。
    unspecified bos;     // シーケンスの先頭を表す表明。
    unspecified eos;     // シーケンスの終端を表す表明。
    unspecified bol;     // 行頭を表す表明。
    unspecified eol;     // 行末を表す表明。
    unspecified bow;     // 語頭を表す表明。
    unspecified eow;     // 語末を表す表明。
    unspecified _b;     // 単語境界を表す表明。
    unspecified _w;     // 単語構成文字にマッチ。
    unspecified _d;     // 数字にマッチ。
    unspecified _s;     // 空白類文字にマッチ。
    proto::terminal< char >::type const _n;     // リテラルの改行 '\n' にマッチ。
```

```
    unspecified _ln;     // 論理改行シーケンスにマッチ。
    unspecified _;     // あらゆる文字にマッチ。
    unspecified self;     // 現在の正規表現オブジェクトへの参照。
    unspecified set;     // 文字セットを作成するのに使用。
    mark_tag const s0;     // Perl の$&部分マッチプレースホルダ。
    mark_tag const s1;     // Perl の$1 部分マッチプレースホルダ。
    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, Expr 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 &);
  }
}
```

mark_tag 構造体

boost::xpressive::mark_tag – 静的正規表現で名前付き捕捉を作成するのに使用する、部分マッチのプレースホルダ型。

書式

```
// ヘッダ：<boost/xpressive/regex_primitives.hpp>

struct mark_tag {
```

```
  // 構築、コピー、解体
  mark_tag(int);

  // 非公開静的メンバ関数
  static unspecified make_tag(int);
};
```

説明

mark_tag は部分マッチのグローバルなプレースホルダ s0、s1、...の型である。mark_tag を使用すると、より意味のある名前で部分マッチプレースホルダを作成できる。動的正規表現における「名前付き捕捉」機能とおおよそ等価である。

名前付き部分マッチプレースホルダは、一意な整数で初期化して作成する。この整数はプレースホルダを使用する正規表現内で一意でなければならない。静的正規表現内でこれに部分式を代入して部分マッチを作成するか、すでに作成した部分マッチを後方参照できる。

```
mark_tag number(1); // number は s1 と等価
// 数字、続いて空白、再び同じ数字にマッチ
sregex rx = (number = +_d) >> ' ' >> number;
```

regex_match()か regex_search()が成功した後は、部分マッチのプレースホルダを match_results<>オブジェクトの添字にして、対応する部分マッチを得る。

mark_tag 構築、コピー、解体の公開演算

```
mark_tag(int mark_nbr);
```

mark_tag プレースホルダを初期化する。

引数: *mark_nbr* この mark_tag を使用する静的正規表現内でこの mark_tag を一意に識別する整数。

要件: mark_nbr > 0

mark_tag 非公開静的メンバ関数

```
static unspecified make_tag(int mark_nbr);
```

（訳注）以下の実体を省略します。

実体	概要	説明
inf グローバル定数	部分式の無限回の繰り返し。	制限なしの繰り返しを指定する repeat<>() 関数テンプレートとともに使用する魔法数。repeat<17, inf>('a')のように使用する。これは Perl の/a{17,}/と等価である。

nil グローバル定数	空のマッチ。	ゼロ幅シーケンスに対するマッチ。nil は常に成功し、文字を消費しない。
alnum グローバル変数	英数字にマッチする。	文字が英数字か決定する正規表現特性。英数字以外にマッチさせる場合は~alnum を使用する。
alpha グローバル変数	アルファベットにマッチする。	文字がアルファベットか決定する正規表現特性。アルファベット以外にマッチさせる場合は~alpha を使用する。
blank グローバル変数	空白（水平空白）にマッチする。	文字が空白か決定する正規表現特性。空白以外にマッチさせる場合は~blank を使用する。
cntrl グローバル変数	制御文字にマッチする。	文字が制御文字か決定する正規表現特性。制御文字以外にマッチさせる場合は~cntrl を使用する。
digit グローバル変数	数字にマッチする。	文字が数字か決定する正規表現特性。数字以外にマッチさせる場合は~digit を使用する。
graph グローバル変数	グラフィカルな文字にマッチする。	文字がグラフィカルな文字か決定する正規表現特性。グラフィカルな文字以外にマッチさせる場合は~graph を使用する。
lower グローバル変数	小文字にマッチする。	文字が小文字か決定する正規表現特性。小文字以外にマッチさせる場合は~lower を使用する。
print グローバル変数	印字可能文字にマッチする。	文字が印字可能文字か決定する正規表現特性。印字可能文字以外にマッチさせる場合は~print を使用する。
punct グローバル変数	区切り文字にマッチする。	文字が区切り文字か決定する正規表現特性。区切り文字以外にマッチさせる場合は~punct を使用する。
space グローバル変数	空白類文字にマッチする。	文字が空白類文字か決定する正規表現特性。空白類文字以外にマッチさせる場合は~space を使用する。
upper グローバル変数	大文字にマッチする。	文字が大文字か決定する正規表現特性。大文字以外にマッチさせる場合は~upper を使用する。
xdigit グローバル変数	16 進数字にマッチする。	文字が 16 進数字か決定する正規表現特性。16 進数字以外にマッチさせる場合は~xdigit を使用する。
bos グローバル変数	シーケンスの先頭を表す表明。	文字シーケンス[begin, end) について、bos はゼロ幅の部分シーケンス[begin, begin) にマッチする。
eos グローバル変数	シーケンスの終端を表す表明。	文字シーケンス[begin, end) について、eos はゼロ幅の部分シーケンス[end, end) にマッチする。
bol グローバル変数	行頭を表す表明。	bol は論理改行シーケンス直後のゼロ幅部分シーケンスにマッチする。この正規表現特性は論理改行シーケンスを規定するのに使用する。
eol グローバル変数	行末を表す表明。	eol は論理改行シーケンス直前のゼロ幅部分シーケンスにマッチする。この正規表現特性は論理改行シー

		ケンスを規定するのに使用する。
bow グローバル変数	語頭を表す表明。	`bow` は非単語構成文字の直後、単語構成文字の直前のゼロ幅部分シーケンスにマッチする。この正規表現特性は単語の構成を規定するのに使用する。
eow グローバル変数	語末を表す表明。	`eow` は単語構成文字の直前、非単語構成文字の直後のゼロ幅部分シーケンスにマッチする。この正規表現特性は単語の構成を規定するのに使用する。
_b グローバル変数	単語境界を表す表明。	`_b` は語頭か語末のゼロ幅部分シーケンスにマッチする。`(bow|eow)` と等価である。この正規表現特性は単語の構成を規定するのに使用する。単語境界以外にマッチさせる場合は`~_b` を使用する。
_w グローバル変数	単語構成文字にマッチする。	`_w` は単語構成文字 1 文字にマッチする。この正規表現特性は単語構成文字を規定するのに使用する。単語構成文字以外にマッチさせる場合は`~_w` を使用する。
_d グローバル変数	数字にマッチする。	`_d` は数字 1 文字にマッチする。この正規表現特性は数字を規定するのに使用する。数字以外にマッチさせる場合は`~_d` を使用する。
_s グローバル変数	空白類文字にマッチする。	`_s` は空白類文字 1 文字にマッチする。この正規表現特性は空白類文字を規定するのに使用する。空白類文字以外にマッチさせる場合は`~_s` を使用する。
_n グローバル定数	リテラルの改行文字`'\n'`にマッチする。	`_n` は改行文字`'\n'`1 文字にマッチする。改行以外の文字にマッチさせる場合は`~_n` を使用する。
_ln グローバル定数	論理改行シーケンスにマッチする。	`_ln` は論理改行シーケンスにマッチする。正規表現特性が規定する行区切りに分類される文字か`'\r\n'`シーケンスである。バックトラックに関して、`'\r\n'`は 1 つの単位として扱う。論理改行以外の 1 文字にマッチさせる場合は`~_ln` を使用する。
_グローバル定数	任意の文字にマッチする。	Perl 構文（/s 修飾子を設定してあるとして）における.と同様、任意の文字にマッチする。_は改行を含む任意の 1 文字にマッチする。
self グローバル変数	現在の正規表現への参照。	再帰正規表現を構築しているときに便利である。識別子 self は現在の正規表現オブジェクトの短縮形である。例えば `sregex rx = '(' >> (self | nil) >> ')';` は、`"((()))"`のような「開きと閉じが正しく対応した括弧群」にマッチする正規表現オブジェクトを作成する。
set グローバル変数	文字集合（文字セット）を作成するのに使用する。	識別子 `set` を使って文字集合を作成する方法は 2 つ

		ある。より簡単なのは `(set= 'a','b','c')` のように集合内の文字をカンマで区切って並べる方法である。この集合は `'a'`、`'b'`、`'c'` にマッチする。もう 1 つは `set` の添字演算子の引数として集合を定義する方法である。例えば `set[ 'a' | range('b','c') | digit ]` は `'a'`、`'b'`、`'c'`、数字にマッチする。 `set` の補集合を得るには~演算子を適用する。例えば `~(set= 'a','b','c')` は `'a'`、`'b'`、`'c'` 以外の文字にマッチする。 `set` は他の集合や補集合と和をとることもできる。例えば `set[ ~digit | ~(set= 'a','b','c') ]` のようにする。
s0 グローバル定数	Perl の `$&` に相当する部分マッチのプレースホルダ。	―
s1 グローバル定数	Perl の `$1` に相当する部分マッチのプレースホルダ。	部分マッチを作成するには、部分式を部分マッチのプレースホルダに代入する。例えば `(s1= _)` は任意の 1 文字にマッチし、どの文字が 1 番目の部分マッチかを記憶する。この部分マッチはパターン内の別の場所から後方参照できる。例えば `(s1= _) >> s1` は任意の文字にマッチし、直後に同じ文字にマッチする。 `regex_match()` や `regex_search()` が成功すると、部分マッチのプレースホルダを `match_results<>` オブジェクトの添字にして N 番目の部分マッチを得られる。
s2 グローバル定数 … s9 グローバル定数	同上。	同上。
a1 グローバル定数 … a9 グローバル定数	―	―
icase 関数テンプレート	部分式を大文字小文字を区別しないようにする。	部分式を大文字小文字を区別しないようにするには `icase()` を使用する。例えば `"foo" >> icase(set['b'] >> "ar")` は `"foo"` の後に `"bar"` が続くシーケンスにマッチするが、後半は大文字小文字を区別しない。
as_xpr 関数テンプレート	リテラルを正規表現にする。	リテラルを正規表現にするには `as_xpr()` を使用する。例えば `"foo" >> "bar"` は右シフト演算子の両方のオペランドが `const char*` であり、そのような演算子

		は存在しないためコンパイルできない。as_xpr("foo") >> "bar"を代わりに使用する。 文字列リテラルだけでなく、文字リテラルに対しても as_xpr()を使用できる。例えば as_xpr('a')は"a"にマッチする。また~as_xpr('a')とすることで文字リテラルの否定が得られる。これは'a'以外の文字にマッチする。
by_ref 関数テンプレート	正規表現オブジェクトを参照により組み込む。	ー
range 関数テンプレート	文字の範囲にマッチする。	範囲[ch_min, ch_max]の任意の文字にマッチする（それぞれ、この関数の引数）。
optional 関数テンプレート	部分式を省略可能にする。!as_xpr(expr)と等価である。	ー
repeat 関数	部分式を複数回繰り返す。	repeat<>()関数テンプレートは 2 形式ある。部分式に *N* 回マッチさせる場合は repeat<N>(expr)を使用する。部分式を *M* から *N* 回マッチさせるには repeat<M,N>(expr)を使用する。 repeat<>()関数は貪欲な数量子を作成する。貪欲でない数量子にするには-repeat<M,N>(expr)のように単項マイナス演算子を適用する。
keep 関数テンプレート	独立部分式を作成する。	部分式のバックトラックを抑止する。部分式内の選択と繰り返しは 1 つの経路だけマッチし、他の選択肢は試行しない。 備考：keep(expr)は Perl の(?>...)拡張と等価である。
before 関数テンプレート	肯定先読み表明。	before(expr)は部分式 *expr* がシーケンス内の現在位置でマッチすれば成功するが、*expr* はマッチに含まれない。例えば before("foo")は現在位置が"foo"の直前であれば成功する。肯定先読み表明はビット否定演算子で否定できる。 備考：before(expr)は Perl の(?=...)拡張と等価である。~before(expr)は否定先読みであり、Perl の(?!...)拡張と等価である。
after 関数テンプレート	肯定後読み表明。	after(expr)は部分式 *expr* がシーケンス内の現在位置から *expr* の長さ戻ったところでマッチすれば成功する。*expr* はマッチに含まれない。例えば after("foo")は現在位置が"foo"の直後であれば成功する。肯定後読み表明はビット否定演算子で否

		定できる。 要件：*expr* の文字数は可変にできない。 備考：`after(expr)` は Perl の `(?<=...)` 拡張と等価である。`~after(expr)` は否定後読みであり、Perl の `(?<!...)` 拡張と等価である。
imbue 関数テンプレート	正規表現特性か `std::locale` を指定する。	`imbue()` は正規表現マッチ時に使用する特性かロカールを正規表現エンジンに対して指示する。特性・ロカールは、正規表現全体で同じものを使用しなければならない。例えば次のコードは正規表現で使用するロカールを指定する：`std::locale loc; sregex rx = imbue(loc)(+digit);`

skip 関数テンプレート

boost::xpressive::skip – 正規表現マッチ中に読み飛ばす文字を指定する。

書式

```
// ヘッダ：<boost/xpressive/regex_primitives.hpp>

template<typename Skip> unspecified skip(Skip const & skip);
```

説明

`skip()` は、正規表現マッチ中に読み飛ばす文字を正規表現エンジンに対して指示する。空白類文字を無視する正規表現を記述するのに最も有用である。例えば、以下は正規表現について空白類文字と区切り文字を読み飛ばすよう指定する。

```
// 文は空白類文字か区切り文字で区切られた
// 1つ以上の単語からなる。
sregex word = +alpha;
sregex sentence = skip(set[_s | punct])( +word );
```

上記の例と同じことをするには正規表現内の各プリミティブの前に `keep(*set[_s | punct])` を挿入する必要がある。「プリミティブ」とは文字列や文字集合、入れ子の正規表現のことである。最後に正規表現の終端にも `*set[_s | punct]` が必要である。上で指定した文の正規表現は以下と等価である。

```
sregex sentence = +( keep(*set[_s | punct]) >> word )
                      >> *set[_s | punct];
```

引数： *skip*　　読み飛ばす文字を指定する正規表現。

備考： 入れ子の正規表現は不可分として扱われるため、読み飛ばしがこれらの処理方法に影響することはない。文字列リテラル

もまた不可分として扱われ、文字列リテラル内で読み飛ばしは発生しない。よって `skip(_s)("this that")` は `skip(_s)("this" >> as_xpr("that"))` と同じではない。前者は"this"と"that"の間に空白が 1 つあるものだけにマッチする。後者は"this"と"that"の間の空白をすべて読み飛ばす。

<boost/xpressive/regex_token_iterator.hpp>ヘッダ

regex_token_iterator の定義と、正規表現を使って文字列をトークンに分割する STL 互換のイテレータがある。

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

regex_token_iterator 構造体テンプレート

boost::xpressive::regex_token_iterator。

書式

```
// ヘッダ：<boost/xpressive/regex_token_iterator.hpp>

template<typename BidiIter>
struct regex_token_iterator {
  // 型
  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;

  // 構築、コピー、解体
  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 &);

  // 公開メンバ関数
```

```
  value_type const & operator*() const;
  value_type const * operator->() const;
  regex_token_iterator< BidiIter > & operator++();
  regex_token_iterator< BidiIter > operator++(int);
};
```

説明

regex_token_iterator 構築、コピー、解体の公開演算

```
regex_token_iterator();
```

事後条件： `*this` がシーケンスイテレータの終端。

```
regex_token_iterator(BidiIter begin, BidiIter end,
                     basic_regex< BidiIter > const & rex);
```

引数： *begin* 検索する文字範囲の先頭。

end 検索する文字範囲の終端。

rex 検索する正規表現パターン。

要件： `[begin,end)` が有効な範囲。

```
template<typename LetExpr>
  regex_token_iterator(BidiIter begin, BidiIter end,
                       basic_regex< BidiIter > const & rex, unspecified args);
```

引数： *begin* 検索する文字範囲の先頭。

end 検索する文字範囲の終端。

rex 検索する正規表現パターン。

args 意味アクションに対して引数束縛した `let()` 式。

要件： `[begin,end)` が有効な範囲。

```
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);
```

引数： *begin* 検索する文字範囲の先頭。

end 検索する文字範囲の終端。

rex 検索する正規表現パターン。

subs —[14]

flags シーケンスに対して正規表現がどのようにマッチするかを制御する省略可能なマッチフラグ（match_flag_type を見よ）。

要件: [begin,end)が有効な範囲。

subs が-1 以上の整数、あるいは全要素が-1 以上の整数である配列か空でない std::vector<>のいずれか。

```
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);
```

引数: *begin* 検索する文字範囲の先頭。

end 検索する文字範囲の終端。

rex 検索する正規表現パターン。

subs —

args 意味アクションに対して引数束縛した let()式。

flags シーケンスに対して正規表現がどのようにマッチするかを制御する省略可能なマッチフラグ（match_flag_type を見よ）。

要件: [begin,end)が有効な範囲。

subs が-1 以上の整数、あるいは全要素が-1 以上の整数である配列か空でない std::vector<>のいずれか。

```
regex_token_iterator(regex_token_iterator< BidiIter > const & that);
```

事後条件: *this == that

```
regex_token_iterator< BidiIter >& operator=(regex_token_iterator< BidiIter > const & that);
```

事後条件: *this == that

regex_token_iterator 公開メンバ関数

```
value_type const & operator*() const;
```

```
value_type const * operator->() const;
```

```
regex_token_iterator< BidiIter > & operator++();
```

14 訳注 原文では記述がありません。詳細はユーザーガイドの文字列の分割とトークン分割の項を参照してください。次の書式も同様です。

N == -1 であれば*this をシーケンスイテレータの終端と同値に設定する。[15]N+1 < subs.size() であれば N を 1 増やし、結果を((subs[N] == -1) ? value_type(what.prefix().str()) : value_type(what[subs[N]].str()))と同値に設定する。それ以外の場合は、what.prefix().first != what[0].second かつ match_prev_avail 要素がフラグに設定されていなければ設定する。次に regex_search(what[0].second, end, what, *pre, flags)を呼び出したのと同様の振る舞いをする。ただし以下の点については振る舞いが異なる。前回見つかったマッチがゼロ幅（what[0].length() == 0）の場合 what[0].second を先頭とする非ゼロ幅マッチを探索する。これが失敗し、かつ what[0].second != suffix().second である場合に限り what[0].second + 1 を先頭とするマッチ（これもゼロ幅かもしれない）を探索する。そのようなマッチが見つかった場合は N を 0 に設定し、結果を ((subs[N] == -1) ? value_type(what.prefix().str()) : value_type(what[subs[N]].str()))と同値に設定する。マッチが見つからなかった場合は last_end を最後に見つかったマッチの終端に設定し、last_end != end かつ subs[0] == -1 であれば N を-1 に、結果を value_type(last_end, end)と同値に設定する。それ以外の場合は*this をシーケンスの終端イテレータと等値に設定する。

```
regex_token_iterator< BidiIter > operator++(int);
```

<boost/xpressive/regex_traits.hpp>ヘッダ

BOOST_XPRESSIVE_USE_C_TRAITS マクロにしたがって C 正規表現特性か C++正規表現特性のヘッダファイルをインクルードする。

```
namespace boost {
  namespace xpressive {
    struct regex_traits_version_1_tag;
    struct regex_traits_version_2_tag;
    template<typename Traits> struct has_fold_case;
    template<typename Char, typename Impl> struct regex_traits;
  }
}
```

（訳注）以下の実体を省略します。

実体	概要	説明
regex_traits_version_1_tag 構造体	―	特性クラスがバージョン 1 の特性インターフェイスに適合することを示すタグ。
regex_traits_version_2_tag 構造体	―	特性クラスがバージョン 2 の特性インターフェイスに適合することを示すタグ。
has_fold_case 構造体テンプレート	特性クラスが fold_case メンバ関数をもつことを示す特性。	―

15 訳注 N の説明は原文にもありませんが、構築直後は 0 である内部変数です。

regex_traits 構造体テンプレート

boost::xpressive::regex_traits。

書式

```
// ヘッダ：<boost/xpressive/regex_traits.hpp>

template<typename Char, typename Impl>
struct regex_traits {
  // 型
  typedef Impl::locale_type locale_type;

  // 構築、コピー、解体
  regex_traits();
  regex_traits(locale_type const &);
};
```

説明

既定の regex_traits 実装（cpp_regex_traits か c_regex_traits）のラッパ。

regex_traits 構築、コピー、解体の公開演算

```
regex_traits();
```

```
regex_traits(locale_type const &);
```

<boost/xpressive/sub_match.hpp>ヘッダ

sub_match<>クラステンプレートと、関連するヘルパ関数の定義がある。

```
namespace boost {
  namespace xpressive {
    template<typename BidiIter> struct sub_match;
    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);
  }
}
```

sub_match 構造体テンプレート

書式

```
// ヘッダ：<boost/xpressive/sub_match.hpp>

template<typename BidiIter>
struct sub_match : public std::pair< BidiIter, BidiIter > {
  // 型
  typedef iterator_value< BidiIter >::type       value_type;
  typedef iterator_difference< BidiIter >::type difference_type;
  typedef unspecified                           string_type;
  typedef BidiIter                              iterator;

  // 構築、コピー、解体
  sub_match();
  sub_match(BidiIter, BidiIter, bool = false);

  // 公開メンバ関数
  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;
  bool matched;  // この部分マッチが全体マッチに関与していれば真。
};
```

説明

sub_match<>型のオブジェクトが表すマーク済み部分式が正規表現マッチに関与している場合、メンバ matched は真と評価されメンバ first および second はマッチを形成する文字範囲 [first,second) を表す。それ以外の場合、matched は偽でありメンバ first および second に未定義の値が入る。

sub_match<>型のオブジェクトが 0 番目の部分式、つまりマッチ全体を表す場合、メンバ matched は常に真である。ただし、フラグ match_partial を正規表現アルゴリズムを渡して部分マッチが得られた場合は例外である。この場合メンバ matched は偽であり、メンバ first および second は部分マッチを形成する文字範囲を表す。

sub_match 構築、コピー、解体の公開演算

```
sub_match();
```

```
sub_match(BidiIter first, BidiIter second, bool matched_ = false);
```

sub_match 公開メンバ関数

```
string_type str() const;
```

```
operator string_type() const;
```

```
difference_type length() const;
```

```
operator bool_type() const;
```

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

```
int compare(string_type const & str) const;
```

字句的な文字列比較を行う。

引数: *str* 比較する文字列。

戻り値: `(*this).str().compare(str)`の結果

```
int compare(sub_match const & sub) const;
```

利便性のためのメンバ関数多重定義。上記関数と受け取る引数が異なるのみ。

```
int compare(value_type const * ptr) const;
```

利便性のためのメンバ関数多重定義。上記関数と受け取る引数が異なるのみ。

<boost/xpressive/traits/c_regex_traits.hpp>ヘッダ

静的・動的正規表現の振る舞いをカスタマイズするCロカール関数のラッパである`c_regex_traits`テンプレートの定義がある。

```
namespace boost {
  namespace xpressive {
    template<typename Char> struct c_regex_traits;

    template<> struct has_fold_case<c_regex_traits< char > >;
  }
}
```

c_regex_traits 構造体テンプレート

boost::xpressive::c_regex_traits – `basic_regex<>`クラステンプレートで使用するために標準の C ロカール関数をカプセル化する。

書式

```
// ヘッダ：<boost/xpressive/traits/c_regex_traits.hpp>

template<typename Char>
struct c_regex_traits {
  // 型
  typedef Char                              char_type;
  typedef std::basic_string< char_type >    string_type;
  typedef unspecified                       locale_type;
  typedef unspecified                       char_class_type;
  typedef regex_traits_version_2_tag        version_tag;
  typedef unspecified                       base_type;

  // 構築、コピー、解体
  c_regex_traits(locale_type const & = locale_type());

  // 公開メンバ関数
  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);

  // 公開静的メンバ関数
  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();
};
```

説明

c_regex_traits 構築、コピー、解体の公開演算

```
c_regex_traits(locale_type const & loc = locale_type());
```

グローバルな C ロカールを使用する c_regex_traits オブジェクトを初期化する。

c_regex_traits 公開メンバ関数

```
bool operator==(c_regex_traits< char_type > const &) const;
```

2 つの c_regex_traits オブジェクトが等値か調べる。

戻り値: 真。

```
bool operator!=(c_regex_traits< char_type > const &) const;
```

2 つの c_regex_traits オブジェクトが等値でないか調べる。

戻り値: 偽。

```
string_type fold_case(char_type ch) const;
```

渡した文字と大文字小文字を区別せずに比較すると等値となる文字をすべて含む string_type を返す。この関数が呼び出されるのは has_fold_case<c_regex_traits<Char> >が真の場合のみである。

引数: *ch* 元の文字。
戻り値: ch と大文字小文字を区別せずに比較すると等値となる文字をすべて含む string_type。

```
locale_type imbue(locale_type loc);
```

何もしない。

```
template<> char widen(char ch);
```

```
template<> wchar_t widen(char ch);
```

```
template<> unsigned char hash(char ch);
```

```
template<> unsigned char hash(wchar_t ch);
```

```
template<> int value(char ch, int radix);
```

```
template<> int value(wchar_t ch, int radix);
```

c_regex_traits 公開静的メンバ関数

```
static char_type widen(char ch);
```

char 型の値を Char 型に変換する。

引数: *ch* 元の文字

戻り値: Char が char であれば *ch*、Char が wchar_t であれば std::btowc(ch)。

```
static unsigned char hash(char_type ch);
```

[0, UCHAR_MAX]の範囲で Char のハッシュ値を返す。

引数: *ch* 元の文字。

戻り値: 0 以上 UCHAR_MAX 以下の値。

```
static char_type translate(char_type ch);
```

何もしない。

引数: *ch* 元の文字。

戻り値: ch

```
static char_type translate_nocase(char_type ch);
```

現在のグローバルな C ロカールを使用して、文字を小文字に変換する。

引数: *ch* 元の文字。

戻り値: Char が char であれば std::tolower(ch)、Char が wchar_t であれば std::towlower(ch)。

```
static char_type tolower(char_type ch);
```

現在のグローバルな C ロカールを使用して、文字を小文字に変換する。

引数: *ch* 元の文字。

戻り値: Char が char であれば std::tolower(ch)、Char が wchar_t であれば std::towlower(ch)。

```
static char_type toupper(char_type ch);
```

現在のグローバルな C ロカールを使用して、文字を大文字に変換する。

引数: *ch* 元の文字。

戻り値: Char が char であれば std::toupper(ch)、Char が wchar_t であれば std::towupper(ch)。

```
static bool in_range(char_type first, char_type last, char_type ch);
```

文字が文字範囲に含まれるか調べる。

引数: *first* 範囲の下限。

last 範囲の上限。

ch 元の文字。

戻り値: first <= ch && ch <= last

```
static bool in_range_nocase(char_type first, char_type last, char_type ch);
```

文字が文字範囲に含まれるか調べる。大文字小文字を区別しない。

引数: *first* 範囲の下限。

last 範囲の上限。

ch 元の文字。

戻り値: in_range(first, last, ch) || in_range(first, last, tolower(ch)) || in_range(first, last, toupper(ch))

備考: 既定の実装は適正な Unicode ケースフォールディングを行わないが、標準 C ロカールではこれが最善である。

```
template<typename FwdIter>
  static string_type transform(FwdIter begin, FwdIter end);
```

イテレータ範囲[F1, F2)が示す文字シーケンスのソートキーを返す。文字シーケンス[G1, G2)が文字シーケンス[H1, H2)の前にソートされる場合に v.transform(G1, G2) < v.transform(H1, H2)とならなければならない。

備考: 現在使用していない。

```
template<typename FwdIter>
  static string_type transform_primary(FwdIter begin, FwdIter end);
```

イテレータ範囲[F1, F2)が示す文字シーケンスのソートキーを返す。大文字小文字を区別せずにソートして文字シーケンス[G1, G2)が文字シーケンス[H1, H2)の前に現れる場合に v.transform(G1, G2) < v.transform(H1, H2)とならなければ

ならない。

備考: 現在使用していない。

```
template<typename FwdIter>
  static string_type lookup_collatename(FwdIter begin, FwdIter end);
```

イテレータ範囲[F1, F2)が示す文字シーケンスが構成する照合要素を表す文字シーケンスを返す。文字シーケンスが正しい照合要素でなければ空文字列を返す。

備考: 現在使用していない。

```
template<typename FwdIter>
  static char_class_type
  lookup_classname(FwdIter begin, FwdIter end, bool icase);
```

指定した文字シーケンスが表す文字分類について、相当するビットマスクを返す。

引数: *begin* 文字分類の名前を表す文字シーケンスの先頭を指す前進イテレータ。
end 文字シーケンスの終端。
icase 戻り値のビットマスクが大文字小文字を区別しない文字分類を表すかを指定する。

戻り値: 文字分類を表すビットマスク。

```
static bool isctype(char_type ch, char_class_type mask);
```

文字分類ビットマスクに対して文字をテストする。

引数: *ch* テストする文字。
mask テストする文字分類のビットマスク。

要件: *mask* は lookup_classname が返したビットマスクか、それらのビット和。

戻り値: 文字が指定した文字分類に含まれれば真、それ以外は偽。

```
static int value(char_type ch, int radix);
```

数字を数値に変換する。

引数: *ch* 数字。
radix 変換に使用する序数。

要件: *radix* は 8、10、16 のいずれか。

戻り値: *ch* が数字でなければ-1、それ以外は文字が表す数値。char_type が char であれば std::strtol を、char_type が wchar_t であれば std::wcstol を使用する。

```
static locale_type getloc();
```

何もしない。

<boost/xpressive/traits/cpp_regex_traits.hpp>ヘッダ

静的・動的正規表現の振る舞いをカスタマイズする std::locale のラッパである cpp_regex_traits テンプレートの定義がある。

```
namespace boost {
  namespace xpressive {
    template<typename Char> struct cpp_regex_traits;

    template<> struct has_fold_case<cpp_regex_traits< char > >;
  }
}
```

cpp_regex_traits 構造体テンプレート

boost::xpressive::cpp_regex_traits – basic_regex<>クラステンプレートで使用するために std::locale をカプセル化する。

書式

```
// ヘッダ：<boost/xpressive/traits/cpp_regex_traits.hpp>

template<typename Char>
struct cpp_regex_traits {
  // 型
  typedef Char                              char_type;
  typedef std::basic_string< char_type >    string_type;
  typedef std::locale                       locale_type;
  typedef unspecified                       char_class_type;
  typedef regex_traits_version_2_tag        version_tag;
  typedef unspecified                       base_type;

  // 構築、コピー、解体
  cpp_regex_traits(locale_type const & = locale_type());

  // 公開メンバ関数
  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(FwdIter, FwdIter) 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);

  // 公開静的メンバ関数
  static unsigned char hash(char_type);
  static char_type translate(char_type);
  static bool in_range(char_type, char_type, char_type);
};
```

説明

cpp_regex_traits 構築、コピー、解体の公開演算

```
cpp_regex_traits(locale_type const & loc = locale_type());
```

指定した std::locale を使用する cpp_regex_traits オブジェクトを初期化する。引数を省略した場合はグローバルな std::locale を使用する。

cpp_regex_traits 公開メンバ関数

```
bool operator==(cpp_regex_traits< char_type > const & that) const;
```

2 つの cpp_regex_traits オブジェクトが等値か調べる。

戻り値: this->getloc() == that.getloc()

```
bool operator!=(cpp_regex_traits< char_type > const & that) const;
```

2 つの cpp_regex_traits オブジェクトが等値でないか調べる。

戻り値: this->getloc() != that.getloc()

```
char_type widen(char ch) const;
```

char 型の値を Char 型に変換する。

引数: *ch* 元の文字。

戻り値: std::use_facet<std::ctype<char_type> >(this->getloc()).widen(ch)

```
char_type translate_nocase(char_type ch) const;
```

内部保持した std::locale を使用して、文字を小文字に変換する。

引数: *ch* 元の文字。

戻り値: std::tolower(ch, this->getloc())

```
char_type tolower(char_type ch) const;
```

内部保持した std::locale を使用して、文字を小文字に変換する。

引数: *ch* 元の文字。

戻り値: std::tolower(ch, this->getloc())

```
char_type toupper(char_type ch) const;
```

内部保持した std::locale を使用して、文字を大文字に変換する。

引数: *ch* 元の文字。

戻り値: std::toupper(ch, this->getloc())

```
string_type fold_case(char_type ch) const;
```

渡した文字と大文字小文字を区別せずに比較すると等値となる文字をすべて含む string_type を返す。この関数が呼び出されるのは has_fold_case<cpp_regex_traits<Char> >が真の場合のみである。

引数: *ch* 元の文字。

戻り値: ch と大文字小文字を区別せずに比較すると等値となる文字をすべて含む string_type

```
bool in_range_nocase(char_type first, char_type last, char_type ch) const;
```

文字が文字範囲に含まれるか調べる。大文字小文字を区別しない。

引数: *first* 範囲の下限。

last 範囲の上限。

ch 元の文字。

戻り値: in_range(first, last, ch) || in_range(first, last, tolower(ch, this->getloc())) || in_range(first, last, toupper(ch, this->getloc()))

備考: 既定の実装は適正な Unicode ケースフォールディングを行わないが、標準の ctype ファセットではこれが最善である。

```
template<typename FwdIter>
  string_type transform(FwdIter begin, FwdIter end) const;
```

現在使用していない。[16]

```
template<typename FwdIter>
  string_type transform_primary(FwdIter begin, FwdIter end) const;
```

イテレータ範囲[F1, F2)が示す文字シーケンスのソートキーを返す。大文字小文字を区別せずにソートして文字シーケンス[G1, G2)が文字シーケンス[H1, H2)の前に現れる場合にv.transform(G1, G2) < v.transform(H1, H2)とならなければならない。

備考: 現在使用していない。

```
template<typename FwdIter>
  string_type lookup_collatename(FwdIter begin, FwdIter end) const;
```

イテレータ範囲[F1, F2)が示す文字シーケンスが構成する照合要素を表す文字シーケンスを返す。文字シーケンスが正しい照合要素でなければ空文字列を返す。

備考: 現在使用していない。

```
template<typename FwdIter>
  char_class_type
  lookup_classname(FwdIter begin, FwdIter end, bool icase) const;
```

指定した文字シーケンスが表す文字分類について、相当するビットマスクを返す。

引数: *begin* 文字分類の名前を表す文字シーケンスの先頭を指す前進イテレータ。
end 文字シーケンスの終端。
icase 戻り値のビットマスクが大文字小文字を区別しない文字分類を表すかを指定する。

戻り値: 文字分類を表すビットマスク。

```
bool isctype(char_type ch, char_class_type mask) const;
```

文字分類ビットマスクに対して文字をテストする。

引数: *ch* テストする文字。
mask テストする文字分類のビットマスク。

要件: *mask*は`lookup_classname`が返したビットマスクか、それらのビット和。

戻り値: 文字が指定した文字分類に含まれれば真、それ以外は偽。

```
int value(char_type ch, int radix) const;
```

数字を数値に変換する。

16 訳注 このメンバ関数の記述は原文にはありません。

引数: *ch* 数字。
radix 変換に使用する序数。

要件: *radix* は 8、10、16 のいずれか。

戻り値: *ch* が数字でなければ-1、それ以外は文字が表す数値。変換は次の要領で行う: `std::stringstream` に `this->getloc()` を指示する。序数を 8、16、10 のいずれかに設定する。*ch* をストリームに挿入する。`int` を抽出する。

```
locale_type imbue(locale_type loc);
```

`*this` に `loc` を指示する。

引数: *loc* `std::locale`。

戻り値: `*this` が直前まで使用していた `std::locale`。

```
locale_type getloc() const;
```

`*this` が現在使用している `std::locale` を返す。

```
template<> unsigned char hash(unsigned char ch);
```

```
template<> unsigned char hash(char ch);
```

```
template<> unsigned char hash(signed char ch);
```

```
template<> unsigned char hash(wchar_t ch);
```

cpp_regex_traits 公開静的メンバ関数

```
static unsigned char hash(char_type ch);
```

`[0, UCHAR_MAX]`の範囲で `Char` のハッシュ値を返す。

引数: *ch* 元の文字。

戻り値: 0 以上 `UCHAR_MAX` 以下の値。

```
static char_type translate(char_type ch);
```

何もしない。

引数: *ch* 元の文字。

戻り値: ch

```
static bool in_range(char_type first, char_type last, char_type ch);
```

文字が文字範囲に含まれるか調べる。

引数: *first* 範囲の下限。
last 範囲の上限。
ch 元の文字。

戻り値: first <= ch && ch <= last

<boost/xpressive/traits/null_regex_traits.hpp>ヘッダ

非文字データを検索する静的・動的正規表現で使用する控えの正規表現特性の実装である、null_regex_traits テンプレートの定義がある。

```
namespace boost {
  namespace xpressive {
    template<typename Elem> struct null_regex_traits;
  }
}
```

null_regex_traits 構造体テンプレート

boost::xpressive::null_regex_traits – 非文字データのための控えの regex_traits。

書式

```
// ヘッダ：<boost/xpressive/traits/null_regex_traits.hpp>

template<typename Elem>
struct null_regex_traits {
  // 型
  typedef Elem                        char_type;
  typedef std::vector< char_type >    string_type;
  typedef unspecified                 locale_type;
  typedef int                         char_class_type;
  typedef regex_traits_version_1_tag version_tag;

  // 構築、コピー、解体
  null_regex_traits(locale_type = locale_type());

  // 公開メンバ関数
  bool operator==(null_regex_traits< char_type > const &) const;
  bool operator!=(null_regex_traits< char_type > const &) const;
  char_type widen(char) const;
```

```
  // 公開静的メンバ関数
  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();
};
```

説明

null_regex_traits 構築、コピー、解体の公開演算

```
null_regex_traits(locale_type = locale_type());
```

null_regex_traits オブジェクトを初期化する。

null_regex_traits 公開メンバ関数

```
bool operator==(null_regex_traits< char_type > const & that) const;
```

2 つの null_regex_traits オブジェクトが等値か調べる。

戻り値: 真。

```
bool operator!=(null_regex_traits< char_type > const & that) const;
```

2 つの null_regex_traits オブジェクトが等値でないか調べる。

戻り値: 偽。

```
char_type widen(char ch) const;
```

char 型の値を Elem 型に変換する。

引数: *ch* 元の文字。

戻り値: `Elem(ch)`

null_regex_traits 公開静的メンバ関数

```
static unsigned char hash(char_type ch);
```

`[0, UCHAR_MAX]`の範囲で`Elem`のハッシュ値を返す。

引数: *ch* 元の文字。

戻り値: 0 以上 `UCHAR_MAX` 以下の値。

```
static char_type translate(char_type ch);
```

何もしない。

引数: *ch* 元の文字。

戻り値: `ch`

```
static char_type translate_nocase(char_type ch);
```

何もしない。

引数: *ch* 元の文字。

戻り値: `ch`

```
static bool in_range(char_type first, char_type last, char_type ch);
```

文字が文字範囲に含まれるか調べる。

引数: *first* 範囲の下限。

last 範囲の上限。

ch 元の文字。

戻り値: `first <= ch && ch <= last`

```
static bool in_range_nocase(char_type first, char_type last, char_type ch);
```

文字が文字範囲に含まれるか調べる。

引数: *first* 範囲の下限。

last 範囲の上限。

ch 元の文字。

戻り値: `first <= ch && ch <= last`

備考: null_regex_traits はケースフォールディングを行わないので、この関数は in_range() と等価である。

```
template<typename FwdIter>
  static string_type transform(FwdIter begin, FwdIter end);
```

イテレータ範囲 [F1, F2) が示す文字シーケンスのソートキーを返す。文字シーケンス [G1, G2) が文字シーケンス [H1, H2) の前にソートされる場合に v.transform(G1, G2) < v.transform(H1, H2) とならなければならない。

備考: 現在使用していない。

```
template<typename FwdIter>
  static string_type transform_primary(FwdIter begin, FwdIter end);
```

イテレータ範囲 [F1, F2) が示す文字シーケンスのソートキーを返す。大文字小文字を区別せずにソートして文字シーケンス [G1, G2) が文字シーケンス [H1, H2) の前に現れる場合に v.transform(G1, G2) < v.transform(H1, H2) とならなければならない。

備考: 現在使用していない。

```
template<typename FwdIter>
  static string_type lookup_collatename(FwdIter begin, FwdIter end);
```

イテレータ範囲 [F1, F2) が示す文字シーケンスが構成する照合要素を表す文字シーケンスを返す。文字シーケンスが正しい照合要素でなければ空文字列を返す。

備考: 現在使用していない。

```
template<typename FwdIter>
  static char_class_type
  lookup_classname(FwdIter begin, FwdIter end, bool icase);
```

null_regex_traits は文字分類をもたないので、lookup_classname() は使用しない。

引数: *begin* 使用しない。

end 使用しない。

icase 使用しない。

戻り値: static_cast<char_class_type>(0)

```
static bool isctype(char_type ch, char_class_type mask);
```

null_regex_traits は文字分類をもたないので、isctype() は使用しない。

引数: *ch* 使用しない。

mask 使用しない。

戻り値: 偽。

```
static int value(char_type ch, int radix);
```

null_regex_traits は数字を解釈しないので、value()は使用しない。

引数: *ch* 使用しない。

radix 使用しない。

戻り値: -1

```
static locale_type imbue(locale_type loc);
```

使用しない。

引数: *loc* 使用しない。

戻り値: loc

```
static locale_type getloc();
```

locale_type()を返す。

戻り値: locale_type()

<boost/xpressive/xpressive.hpp>ヘッダ

静的・動的両方の正規表現サポートを含む xpressive のすべてをインクルードする。

<boost/xpressive/xpressive_dynamic.hpp>ヘッダ

動的正規表現の記述と使用に必要なすべてをインクルードする。

<boost/xpressive/xpressive_fwd.hpp>ヘッダ

xpressive のすべての公開データ型の前方宣言。

```
BOOST_PROTO_FUSION_V2
BOOST_XPRESSIVE_HAS_MS_STACK_GUARD
```

```
namespace boost {
  namespace xpressive {
    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;
  }
}
```

（訳注）以下の実体は省略しました。

- マクロ：BOOST_PROTO_FUSION_V2、BOOST_XPRESSIVE_HAS_MS_STACK_GUARD

<boost/xpressive/xpressive_static.hpp>ヘッダ

静的正規表現の記述と使用に必要なすべてをインクルードする。

<boost/xpressive/xpressive_typeof.hpp>ヘッダ

xpressive を Boost.Typeof ライブラリとともに使用するための型登録。

謝辞

xpressive の開発初期段階で専門的なアドバイスをくれた Joel de Guzman と Hartmut Kaiser に感謝したい。意味アクションの構文をはじめとする、静的 xpressive の構文の大部分は Spirit から借用した。正規表現が標準ライブラリに入ることになった John Maddock の草案について、その優れた業績に感謝したい。彼の正規表現実装からは様々なアイデアを借用した。入れ子の正規表現の振舞いに関する助言をくれた Andrei Alexandrescu と正規表現ドメイン固有言語の示唆をくれた Dave Abrahams にも感謝したい。Noel Belcourt は xpressive を Metroworks CodeWarrior コンパイラに移植するのを手伝ってくれた。Markus Schöpflin は HP Tru64 におけるバグの追跡を手伝ってくれ、Steven Watanabe はその修正方法について示唆をくれた。

xpressive の意味アクションのコードと文書、記号表と属性の両方についてアイデアを提供してくれた David Jenkins にも感謝したい。数値解析の例である `libs/xpressive/example/numbers.cpp` にある xpressive の三分検索木の実装と記号表と属性のドキュメントは David によるものである。

最後に、xpressive のレビューマネージャを務めてくれた Thomas Witt に感謝したい。

付録

付録 1：履歴[17]

Boost 1.55.0（2013 年 11 月 11 日）

- 不完全な文字集合について assert ではなく例外を投げるようにした（チケット#8843）。
- 未使用のローカルな typedef を削除した（チケット#8880）。
- `sequence_stack.hpp` で try/catch の代わりに RAII を使用するようにした（チケット#8882）。
- clang の `-Wimplicit-fallthrough` 診断が正しく動作するようにした（チケット#8474）。

Boost 1.54.0（2013 年 7 月 1 日）

- 未使用の変数を削除した。チケット#8039 を修正した。
- glx.h のマクロ None と名前が衝突していたのを回避した。チケット#8204 を修正した。
- gcc で出ていた警告に対応した。チケット#8138 を修正した。

Boost 1.53.0（2013 年 2 月 4 日）

- 最近の Boost 版スマートポイントの変更に対応した。チケット#7809 を修正した。

Boost 1.52.0（2012 年 11 月 5 日）

- sub_match で Boost.Range が使えるようにした。チケット#7237 を修正した。

Boost 1.50.0（2012 年 6 月 28 日）

- たちの悪い `lexical_cast` ハックを幾分マシなものにした。
- C++11 で問題になる MPL の assert を static assert に置き換えた。チケット#6846 を修正した。

Boost 1.49.0（2012 年 2 月 24 日）

- gcc で未使用として警告が出ていた変数を削除した。

Boost 1.45.0（2010 年 11 月 19 日）

- `xpressive::as` がワイド文字版の `sub_match` オブジェクトを処理できなかったバグを修正（チケット#4496）。

Boost 1.44.0（2010 年 8 月 13 日）

- `nested_results` における、`typedef` を使った移植性のない `using` ディレクティブを置き換えた。

17 訳注　原文ではバージョン 2.1.0 までしか記述がありませんが、翻訳版では Boost のリリースノートから以降の履歴を抜粋しました。2.1.0 が Boost 1.36（2008 年 8 月 14 日）に相当します。

- 非ローカル変数に対するプレースホルダを使ったユーザー定義表明をサポートした。

Boost 1.43.0（2010 年 5 月 6 日）

- <boost/xpressive/regex_error.hpp>へのインクルードが欠けていたのを修正。

Boost 1.42.0（2010 年 2 月 2 日）

- match_results を std::list の未定義の動作に依存しないようにした（チケット#3278）。
- 「既定コンストラクタで構築したイテレータをコピーしてはならない。」（チケット#3538）
- GCC および Darwin で警告が出ないようにした（チケット#3734）。

Boost 1.41.0（2009 年 11 月 17 日）

- \Q～\E 引用メタを使用した場合に無限ループする場合があるバグを修正（チケット#3586）。
- MSVC で到達不能コードの警告を出ないようにした。
- MSVC で/Za（「言語拡張を無効にする」）モードにした場合に発生する警告とエラーを解決した。
- 様々なコンパイラの C++0x モードに関するバグを修正した。

Boost 1.40.0（2009 年 8 月 27 日）

- Visual C++ 10.0 で動作するようになった（チケット#3124）。

Boost 1.39.0（2009 年 5 月 2 日）

- GCC の最適化で純粋仮想関数呼び出しによる実行時エラーが発生する問題（チケット#2655）の回避策を追加。

Boost 1.38.0（2009 年 2 月 8 日）

- std::basic_regex との互換性のために、basic_regex に入れ子の syntax_option_flags と value_type 型定義を追加。
- Proto v4 への移植。boost/xpressive/proto にあった Proto v2 は削除した。
- regex_error を boost::exception から継承するようにした。

バージョン 2.1.0（2008 年 6 月 12 日）

新機能：

- 静的正規表現に skip() プリミティブを追加。入力文字列内の、正規表現マッチ中に無視する部分を指定できる。
- regex_replace() アルゴリズムの範囲ベースのインターフェイス。
- regex_replace() が書式化文字列に加えて、書式化オブジェクトと書式化ラムダ式を受け付けるようになった。

バグ修正：

- 前方先読み、後方先読み、独立部分式における意味アクションがクラッシュせず、積極実行されるようになった。

バージョン 2.0.1（2007 年 10 月 23 日）

バグ修正：

- `sub_match<>`コンストラクタが、既定コンストラクタで構築したイテレータをコピーするとデバッグ表明にヒットする。

バージョン 2.0.0（2007 年 10 月 12 日）

新機能：

- 意味アクション。
- カスタムの表明。
- 名前付き捕捉。
- 動的正規表現文法。
- `(?R)`構造による動的再帰正規表現。
- 非文字データ検索のサポート。
- 不正な静的正規表現に対するエラーを改善した。
- 正規表現アルゴリズムの範囲ベースのインターフェイス。
- `match_flag_type::format_perl`、`match_flag_type::format_sed`および`match_flag_type::format_all`。
- `operator+(std::string, sub_match<>)`とその変種。
- バージョン 2。このバージョンの正規表現特性は`tolower()`と`toupper()`をもつ。

バグ修正：

- `~(set='a')`のような 1 文字の否定が動作するようになった。

バージョン 1.0.2（2007 年 4 月 27 日）

バグ修正：

- 事前の予告どおり、10 番目以降の後方参照が動作するようになった。

このバージョンは Boost 1.34 とともにリリースされた。

バージョン 1.0.1（2006 年 10 月 2 日）

バグ修正：

- `match_results::position()`が入れ子の結果に対して動作するようになった。

バージョン 1.0.0（2006 年 3 月 16 日）

バージョン 1.0！

バージョン 0.9.6（2005 年 6 月 19 日）

Boost の受理に向けてレビューされたバージョン。レビューが始まったのは 2005 年 11 月 8 日。2005 年 11 月 28 日、xpressive は Boost に受理された。

バージョン 0.9.3（2005 年 6 月 30 日）

新機能：

- TR1 形式の `regex_traits` インターフェイス。
- 高速化。
- `syntax_option_type::ignore_white_space`。

バージョン 0.9.0（2004 年 9 月 2 日）

新機能：

- いろいろ。

バージョン 0.0.1（2003 年 11 月 16 日）

xpressive のアナウンス：http://lists.boost.org/Archives/boost/2003/11/56312.php

付録 2：未実装の項目

以下の機能は xpressive 2.x で実装を予定している。

- `syntax_option_type::collate`
- `[.a.]`のような照合シーケンス
- `[=a=]`のような等価クラス
- `syntax_option_type::nosubs` を使用したときの入れ子結果の生成制御。静的正規表現における `nosubs()` 修飾子。

ウィッシュリスト。あなたやあなたの会社がお金をくれるなら考えてもいいよ！

- 単純な正規表現を高速化する、最適化 DFA バックエンド。
- basic、extended、awk、grep および egrep 正規表現構文のための別の正規表現コンパイラフロントエンド。
- 動的正規表現構文のより細かい制御。
- ICU を使った完全な Unicode サポート。

- 地域化サポートの強化（可能な限り std::locale のカスタムファセットを使う）。

付録 3：Boost.Regex との違い

xpressive のユーザーの多くは Boost.Regex ライブラリになじんでいると思うので、xpressive と Boost.Regex の重大な違いについて見落としているとしたら私の怠慢である。特に以下の点を挙げる。

- xpressive::basic_regex<>は、文字型ではなくイテレータ型に対するテンプレートである。
- xpressive::basic_regex<>は、文字列から直接構築できない。文字列から正規表現オブジェクトを構築するには、代わりに basic_regex::compile()か regex_compiler<>を使用しなければならない。
- xpressive::basic_regex<>は imbue()メンバ関数を持たない。代わりに xpressive::compiler<>ファクトリに imbue()メンバ関数がある。
- boost::basic_regex<>は std::basic_string<>メンバのサブセットをもつが、xpressive::basic_regex<>にはない。欠けているメンバは、assign()、operator[]()、max_size()、begin()、end()、size()、compare()および operator=(std::basic_string<>)である。
- 他に boost::basic_regex<>にあって xpressive::basic_regex<>にないメンバ関数は、set_expression()、get_allocator()、imbue()、getloc()、getflags()および str()である。
- xpressive::basic_regex<>は RegexTraits テンプレート引数をもたない。正規表現構文と地域化に関する振る舞いをカスタマイズするには、regex_compiler<>およびカスタムの std::locale 正規表現ファセットを使用する。
- xpressive::basic_regex<>および xpressive::match_results<>は Allocator テンプレート引数を持たない。これはこういう設計である。
- match_not_dot_null および match_not_dot_newline は match_flag_type 列挙から syntax_option_type 列挙に移動しており、名前も not_dot_null および not_dot_newline に変更している。
- 次の syntax_option_type 列挙値はサポートしない：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。
- 次の match_flag_type 列挙値はサポートしない：match_not_bob、match_not_eob、match_perl、match_posix および match_extra。

また、現在の実装では xpressive の正規表現アルゴリズムは病的な振る舞いや例外によるアボートを検出しない。病的な振る舞いをせず効率のよいパターンを書くのはあなたの責任である。

付録 4：パフォーマンスの比較

xpressive の効率は Boost.Regex に負けず劣らずである。パフォーマンスベンチマークを走らせ、静的正規表現、動的正規表現

Boost.Regex を 2 つのプラットフォーム（gcc（Cygwin）、Visual C++）で比較した。テストは短いマッチと長い検索を行った。いずれのプラットフォームでも、短いマッチでは xpressive が高速であり、長い検索では大体 Boost.Regex と同じだった。[18]

xpressive 対 Boost.Regex（GCC（Cygwin））

- 静的正規表現
- 動的正規表現
- Boost.Regex

以上のパフォーマンス比較結果を次に示す。

テスト仕様

ハードウェア：

ハイパースレッディング 3GHz Xeon 1GB RAM

オペレーティングシステム：

Windows XP Professional + Cygwin

コンパイラ：

GNU C++ 3.4.4（Cygwin 版）

C++標準ライブラリ：

GNU libstdc++ 3.4.4

Boost.Regex のバージョン：

1.33+, BOOST_REGEX_USE_CPP_LOCALE, BOOST_REGEX_RECURSIVE

xpressive のバージョン：

0.9.6a

比較 1：短いマッチ

以下のテストでは入力文字列に対する正規表現マッチに要した時間を評価項目とする。各テスト結果の上側の数値は最速時間に対して正規化しており、1.0 が最もよい。下側の（括弧で囲んだ）数値は実際の秒時間である。最良の結果は緑色にしてある。

短いマッチ

静的	動的	Boost	テキスト	正規表現

18 おことわり：すべてのベンチマークについて、真のテストとは**あなたの**パターンと**あなたの**入力に対して**あなたの**プラットフォームで xpressive がどう動くかである。よってあなたのアプリケーションで効率が重要なのであれば、あなた自身でテストをするのが最もよい。

xpressive	xpressive			
1 (8.79e-07s)	1.08 (9.54e-07s)	2.51 (2.2e-06s)	100-といった、メッセージ文字列を含む FTP 応答行。	`^([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}$`
1.12 (9.38e-07s)	1 (8.34e-07s)	2.5 (2.08e-06s)	SW1 1ZZ	`^[a-zA-Z]{1,2}[0-9][0-9A-Za-z]{0,1}{0,1}[0-9][A-Za-z]{2}$`
1 (7.9e-07s)	1.06 (8.34e-07s)	2.49 (1.96e-06s)	4/1/2001	`^[[:digit:]]{1,2}/[[:digit:]]{1,2}/[[:digit:]]{4}$`
1 (8.19e-07s)	1.04 (8.49e-07s)	2.4 (1.97e-06s)	12/12/2001	`^[[:digit:]]{1,2}/[[:digit:]]{1,2}/[[:digit:]]{4}$`
1.09 (8.95e-07s)	1 (8.19e-07s)	2.4 (1.96e-06s)	123	`^[-+]?[[:digit:]]*\.?[[:digit:]]*$`
1.11 (8.79e-07s)	1 (7.9e-07s)	2.57 (2.03e-06s)	+3.14159	`^[-+]?[[:digit:]]*\.?[[:digit:]]*$`
1.09 (8.94e-07s)	1 (8.19e-07s)	2.47 (2.03e-06s)	-3.14159	`^[-+]?[[:digit:]]*\.?[[:digit:]]*$`

比較 2：長い検索

次のテストは長い英文テキストから**すべての**マッチを見つけるのに要した時間を測定した。Project Gutenberg の Mark Twain の完全なテキストを使用した。テキストの長さは 19MB である。上と同様に上側の数値は正規化時間であり、下側の数値は実際の時間である。最短時間は緑色にした。

長い検索

静的 xpressive	動的 xpressive	Boost	正規表現
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|Huckleberry|Finn`
1.56 (0.195s)	1.12 (0.141s)	1 (0.125s)	`(Tom|Sawyer|Huckleberry|Finn).{0,30}river|river.{0,30}(Tom|Sawyer|Huckleberry|Finn)`

xpressive 対 Boost.Regex（Visual C++）

- 静的 xpressive
- 動的 xpressive
- Boost.Regex

以上のパフォーマンス比較結果を次に示す。

テスト仕様

ハードウェア：

ハイパースレッディング 3GHz Xeon 1GB RAM

オペレーティングシステム：

Windows XP Professional

コンパイラ：

Visual C++ .NET 2003（7.1）

C++標準ライブラリ：

Dinkumware バージョン 313

Boost.Regex のバージョン：

1.33+, BOOST_REGEX_USE_CPP_LOCALE, BOOST_REGEX_RECURSIVE

xpressive のバージョン：

0.9.6a

比較 1：短いマッチ

以下のテストでは入力文字列に対する正規表現マッチに要した時間を評価項目とする。各テスト結果の上側の数値は最速時間に対して正規化しており、1.0 が最もよい。下側の（括弧で囲んだ）数値は実際の秒時間である。最良の結果は緑色にしてある。

短いマッチ

静的 xpressive	動的 xpressive	Boost	テキスト	正規表現
1 (3.2e-007s)	1.37 (4.4e-007s)	2.38 (7.6e-007s)	100-といった、メッセージ文字列を含む FTP 応答行。	`^([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 (1.1e-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}$`
1 (3.2e-007s)	1.09 (3.5e-007s)	1.86 (5.96e-007s)	SW1 1ZZ	`^[a-zA-Z]{1,2}[0-9][0-9A-Za-z]{0,1}{0,1}[0-9][A-Za-z]{2}$`
1 (2.68e-007s)	1.22 (3.28e-007s)	2 (5.36e-007s)	4/1/2001	`^[[:digit:]]{1,2}/[[:digit:]]{1,2}/[[:digit:]]{4}$`
1 (2.76e-007s)	1.16 (3.2e-007s)	1.94 (5.36e-007s)	12/12/2001	`^[[:digit:]]{1,2}/[[:digit:]]{1,2}/[[:digit:]]{4}$`
1 (2.98e-007s)	1.03 (3.06e-007s)	1.85 (5.51e-007s)	123	`^[-+]?[[:digit:]]*\.?[[:digit:]]*$`
1 (3.2e-007s)	1.12 (3.58e-007s)	1.81 (5.81e-007s)	+3.14159	`^[-+]?[[:digit:]]*\.?[[:digit:]]*$`
1 (3.28e-007s)	1.11 (3.65e-007s)	1.77 (5.81e-007s)	-3.14159	`^[-+]?[[:digit:]]*\.?[[:digit:]]*$`

比較 2：長い検索

次のテストは長い英文テキストから**すべての**マッチを見つけるのに要した時間を測定した。Project Gutenberg の Mark Twain の完

全なテキストを使用した。テキストの長さは 19MB である。上と同様に上側の数値は正規化時間であり、下側の数値は実際の時間である。最短時間は緑色にした。

長い検索

静的 xpressive	動的 xpressive	Boost	正規表現
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|Huckleberry|Finn`
1.24 (0.164s)	1.44 (0.191s)	1 (0.133s)	`(Tom|Sawyer|Huckleberry|Finn).{0,30}river|river.{0,30}(Tom|Sawyer|Huckleberry|Finn)`

付録 5：実装ノート

tracking_ptr<>による循環型コレクション

xpressive では正規表現オブジェクトがお互いや自分自身を値や参照で参照する場合がある。また参照先の正規表現が生存するために参照カウントを使っている。これにより循環参照が生じ、メモリリークが起きる可能性がある。xpressive は `tracking_ptr<>` という型を使ってリークを回避している。本ドキュメントでは `tracking_ptr<>` の、高水準な観点からの振る舞いについて述べる。

制限

以下に挙げる設計上の制限を満たす解法でなければならない。

- 懸垂参照が発生しない：直接・間接的に参照しているオブジェクトはすべて参照が必要な限りは生存しなければならない。
- メモリリークが発生しない：オブジェクトは、最終的にはすべて確実に解放しなければならない。
- ユーザーの介入がない：ユーザーによる明示的な循環回収ルーチン呼び出しを必要としてはならない。
- クリーンアップ処理が例外を送出しない：回収処理はデストラクタから呼び出される可能性が高いため、どのような事情があろうとも例外を送出してはならない。

ハンドル・ボディイディオム

`tracking_ptr<>` を使うには、型をハンドルとボディに分離しなければならない。xpressive の場合、ハンドル型は `basic_regex<>` でありボディは `regex_impl<>` である。ハンドルがボディへの `tracking_ptr<>` をもつ。

ボディ型は enable_reference_tracking<> を継承しなければならない。これで tracking_ptr<> が使用する帳簿（bookkeeping）となるデータ構造がボディに与えられる。

1. std::set<shared_ptr<body> > refs_ :このボディが参照するボディのコレクション。
2. std::set<weak_ptr<body> > refs_ :このボディを参照するボディのコレクション。

参照と依存

上記 1.を「参照」、2.を「依存」と呼ぶことにする。tracking_ptr<>は直接参照されるオブジェクトと（他の参照を介して）間接的に参照されるオブジェクトの両方を参照の集合として扱う、ということは理解しておかなければならない。依存の集合についても同じことが当てはまる。言い換えると、各ボディはそのボディが必要とする他のあらゆるボディに対する直接の参照カウンタをもつ。

なぜこれが重要なのか？あるボディを参照するハンドルがなくなった時点で、そのすべての参照を懸垂参照を心配せずに即座に解放可能だからである。

参照と依存は相互交流の関係である。動作を以下に示す。

1. オブジェクトが他のオブジェクトを参照として得ると、参照先のオブジェクトは参照元のオブジェクトを依存として得る。
2. これに加えて参照元のオブジェクトは参照先のオブジェクトがもつ参照をすべて得、参照先のオブジェクトは参照元のオブジェクトがもつ依存をすべて得る。
3. オブジェクトが新たな参照を獲得すると、その参照はすべての依存オブジェクトにも追加される。
4. オブジェクトが新たな依存を獲得すると、その依存はすべての参照オブジェクトにも追加される。
5. オブジェクトが自分自身の依存をもつことは認められない。オブジェクトが自分自身を参照することは可能であり、よくある。

次のコードを考える。

```
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)
```

参照と依存がどのように伝播するか 1 行ずつ見ていく。

式	効果
1) sregex group = '(' >> by_ref(expr) >> ')';	group: cnt1 refs{expr} deps={} expr: cnt2 refs{} deps={group}
2) sregex fact = +_d | group;	group: cnt2 refs{expr} deps={fact} expr: cnt3 refs{} deps={group,fact} fact: cnt1 refs{expr,group} deps={}
3) sregex term = fact >> *(('*' >> fact) | ('/' >> fact));	group: cnt3 refs{expr} deps={fact,term} expr: cnt4 refs{} deps={group,fact,term} fact: cnt2 refs{expr,group} deps={term}

	term: cnt1 refs{expr,group,fact} deps={}
4) expr = term >> *(('+' >> term) | ('-' >> term));	group: cnt5 refs{expr,group,fact,term} deps={expr,fact,term} expr: cnt5 refs{expr,group,fact,term} deps={group,fact,term} fact: cnt5 refs{expr,group,fact,term} deps={expr,group,term} term: cnt5 refs{expr,group,fact,term} deps={expr,group,fact}
5) }	expr: cnt2 refs{expr,group,fact,term} deps={group,fact,term}

オブジェクトの循環が発生したときに参照と依存がどのように伝播するかを示している。(4)の行で循環が閉じられ、以降、各オブジェクトは他のオブジェクトに対して参照カウントをもつ。これでなぜリークしないのか？ 先を続けよう。

循環を破る（循環ブレーカ）

ボディは参照と依存の集合をもつ、というところまでは分かった。循環をいつどこで破るかがまだ決まっていない。これはハンドルの一部になっている tracking_ptr<>の仕事である。tracking_ptr<>は 2 つの shared_ptr をもつ。1 つ目は明らかなように shared_ptr<body>であり、このハンドルが参照するボディへの参照である。2 つ目の shared_ptr は循環を破るのに使用し、ボディへのハンドルがすべてスコープから出たときに、ボディがもつ参照の集合を解放する。

このことから分かるように 1 つのボディに対するハンドルは 2 つ以上になる可能性がある。実際、tracking_ptr<>は「書き込み時コピー」セマンティクスを用いており、ハンドルをコピーするとボディは共有される。あるボディへのハンドルは、そのうちすべてスコープの外に出る。このとき、他のボディ（当該ボディそのものかもしれない）が参照を保持していてボディへの参照カウントは 0 より大きいかもしれない。しかし循環ブレーカはハンドル内にしか存在しないので、循環ブレーカの参照カウントは間違いなく 0 である。ハンドルが存在しなければ循環ブレーカも存在しない。

循環ブレーカが行うことは何だろう？ ボディが std::set<shared_ptr<body> >型の参照の集合をもつことを思い出していただきたい。この型を「references_type」と呼ぶことにする。循環ブレーカは shared_ptr<references_type>であり、以下に示すカスタムの削除オブジェクトを使う。

```
template<typename DerivedT>
struct reference_deleter
{
    void operator ()(std::set<shared_ptr<DerivedT> > *refs) const
    {
        refs->clear();
    }
};
```

循環ブレーカの役割は、ボディへの最後のハンドルがなくなったときにボディがもつ参照の集合を解放すること、それだけである。

すべてのボディが最終的に解放されることが保証されるのは明らかである。ハンドルがすべてスコープから出ると、ボディがもつすべての参照が解放され、後には何も（非 0 の参照カウントも）残らない。リークが起こらないことが保証される。

以上のことから懸垂参照が発生しないことが保証される、と言うのは少し難しい。A、B、C の 3 つのボディがあるとしよう。A は B を参照し、B は C を参照する。B へのハンドルがすべてスコープから出ると、B がもつ参照の集合が解放される。A が（間接的に）使用

しているにも関わらず、これではCが削除されてしまうのではないか？ そうはならない。AはBだけでなくCも直接参照し続けるように、上記のように参照と依存を伝播しているため、このような状況は起こらない。Bがもつ参照の集合が解放されても、各ボディはAが使用中のため削除されない。

将来

`std::set`、`shared_ptr`、`weak_ptr`を使っているが、軒並み効率が悪い。手ごろなので使っているだけなのだが。改善できると思う。

また、オブジェクトが必要以上に長い時間生存する場合がある。

```
sregex b;
{
    sregex a = _;
    b = by_ref(a);
    b = _;
}
// a がこの時点でまだ生存している！
```

参照と依存を伝播する手法であるため、`std::set`は拡大するのみである。参照が不要になった場合でも縮小しない。xpressiveではこれは問題にならない。参照オブジェクトのグラフは大きくならず、それぞれ孤立したままである。汎用の参照カウント式循環コレクション機構として`tracking_ptr<>`を使おうとすると、この問題に焦点が当てられることになるだろう。