Boost.Regex 5.0.1

John Maddock

翻訳にあたって

- 本書は Boost.Regex 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 1.58.0 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。

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。

構成

コンパイラセットアップ

Boost.Config サブシステムがあるため、Boost.Regex を使うのに特別な構成は必要ない。問題がある場合（あるいは一般的でないコンパイラやプラットフォームを使う場合）は Boost.Config に構成スクリプトがあるので、そちらを使うとよい。

ロカールおよび特性クラスの選択

ユーザのロカール（locale）を Boost.Regex がどのように処理するか制御するには、以下のマクロ（`user.hpp` を見よ）を使う。

マクロ	説明
`BOOST_REGEX_USE_C_LOCALE`	Boost.Regex が、特性クラス中で大域 C ロカールを使うように強制する。C++ロカールがあるのでこの設定は現在非推奨となっている。
`BOOST_REGEX_USE_CPP_LOCALE`	Boost.Regex が、既定特性クラス中で `std::locale` を使うように強制する。各正規表現はインスタンス固有のロカールにより `imbue` される。これは Windows 以外のプラットフォームにおける既定の動作である。
`BOOST_REGEX_NO_W32`	Boost.Regex は（利用可能な場合でも）あらゆる Win32 API を使用しない（`BOOST_REGEX_USE_C_LOCALE` が設定されない限り `BOOST_REGEX_USE_CPP_LOCALE` が暗黙に有効になる）。

リンケージに関するオプション

マクロ	説明
`BOOST_REGEX_DYN_LINK`	Microsoft C++および Borland C++ビルドでは Boost.Regex の DLL ビルドにリンクするようになる。既定では Boost.Regex は、動的 C 実行時ライブラリを使用している場合であっても静的ライブラリにリンクする。
`BOOST_REGEX_NO_LIB`	Microsoft C++および Borland C++において、Boost.Regex がリンクするライブラリを自動的に選択しないようにする。
`BOOST_REGEX_NO_FASTCALL`	Microsoft ビルドにおいて、`__fastcall` 呼び出し規約よりも `__cdecl` 呼び出し規約を使用する。マネージ・アンマネージコードの両方から同じライブラリを使用する場合に有用。

アルゴリズムの選択

マクロ	説明
`BOOST_REGEX_RECURSIVE`	スタック再帰マッチアルゴリズムを使用する。これは通常最速のオプションである（といっても効果はわずかだが）が、極端な場合はスタックオーバーフローを起こす可能性がある

	（Win32 では安全に処理されるが、その他のプラットフォームではそうではない）。
BOOST_REGEX_NON_RECURSIVE	非スタック再帰マッチアルゴリズムを使用する。スタック再帰に比べて若干遅いが、どれだけ病的な正規表現に対しても安全である。これは Win32 以外のプラットフォームにおける既定である。

アルゴリズムの調整

以下のオプションは BOOST_REGEX_RECURSIVE が設定されている場合のみ有効である。

マクロ	説明
BOOST_REGEX_HAS_MS_STACK_GUARD	Microsoft スタイルの __try - __except ブロックがサポートされており、スタックオーバーフローを安全に捕捉できることを Boost.Regex に通知する。

以下のオプションは BOOST_REGEX_NON_RECURSIVE が設定されている場合のみ有効である。

マクロ	説明
BOOST_REGEX_BLOCKSIZE	非再帰モードにおいて Boost.Regex は状態マシンのスタックのために大きめのメモリブロックを使う。ブロックのサイズが大きいほどメモリ確保の回数は少なくなる。既定は 4096 バイトであり、大抵の正規表現マッチでメモリの再確保が必要ない値である。しかしながらプラットフォームの特性を見た上で、別の値を選択することも可能である。
BOOST_REGEX_MAX_BLOCKS	サイズ BOOST_REGEX_BLOCKSIZE のブロックをいくつ使用できるか設定する。この値を超えると Boost.Regex はマッチの検索を停止し、std::runtime_error を投げる。既定値は 1024 である。BOOST_REGEX_BLOCKSIZE を変更した場合、この値にも微調整が必要である。
BOOST_REGEX_MAX_CACHE_BLOCKS	内部キャッシュに格納するメモリブロック数を設定する。メモリブロックは ::operator new 呼び出しではなくこのキャッシュから割り当てられる。一般的にこの方法はメモリブロック要求のたびに ::operator new を呼び出すよりも数段高速だが、巨大なメモリチャンク（サイズが BOOST_REGEX_BLOCKSIZE のブロックが最大 16 個）をキャッシュしなければならないという欠点がある。メモリの制限が厳しい場合は、この値を 0 に設定し（キャッシュはまったく行われない）、それで遅すぎる場合は 1 か 2 にするとよい。逆に巨大なマルチプロセッサ、マルチスレッドのシステムでは大きな値のほうがよい。

ライブラリのビルドとインストール

ライブラリの zip ファイルを解凍するとき、ディレクトリの内部構造を変更しないようにする（例えば-d オプションを付けて解凍する）。もし変更してしまっていたら、この文書を読むのをやめて解凍したファイルをすべて削除して最初からやり直したほうがよい。

本ライブラリを使用する前に設定することは何もない。大抵のコンパイラ、標準ライブラリ、プラットフォームは何もしなくてもサポートされる。設定で何か問題がある場合や、単にあなたのコンパイラで設定をテストしてみたい場合は、やり方は他の Boost ライブラリと同じである。ライブラリの設定ドキュメントを見るとよい。

ライブラリのコードはすべて名前空間 `boost` 内にある。

他のテンプレートライブラリとは異なり、本ライブラリはテンプレートコード（ヘッダ中）と、静的コード・データ（cpp ファイル中）が混在している。したがってライブラリを使用する前に、ライブラリのサポートコードをビルドしてライブラリかアーカイブファイルを作成する必要がある。これについて各プラットフォームにおける方法を以下に述べる。

bjam を用いたビルド

本ライブラリをビルドおよびインストールする最適な方法である。Getting Started ガイドを参照していただきたい。

Unicode および ICU サポートビルド[2]

Boost.Regex は、ICU がコンパイラの検索パスにインストールされているか設定をチェックするようになった。ビルドを始めると次のようなメッセージが現れるはずである：

```
Performing configuration checks

    - has_icu builds           : yes
```

これは ICU が見つかり、ライブラリのビルドでサポートされるということを表している。

ヒント

正規表現ライブラリで ICU を使用したくない場合は--disable-icu コマンドラインオプションを使用してビルドするとよい。

仮に次のような表示が出た場合、

```
Performing configuration checks

    - has_icu builds           : no
```

ICU は見つからず、関連するサポートはライブラリのコンパイルに含まれない。これが期待した結果と違うという場合は、ファイル `boost-root/bin.v2/config.log` の内容を見て、設定チェック時にビルドが吐き出した実際のエラーメッセージを確認すべきで

2　訳注　Unicode を用いた正規表現ライブラリは ICU にもあります。Unicode に関する機能は ICU 版のほうが豊富です。

ある。コンパイラに適切なオプションを渡してエラーを修正する必要があるだろう。例えば、*some-include-path* をコンパイラのヘッダインクルードパスに追加するには次のようにする。

```
bjam include=some-include-path --toolset=toolset-name install
```

あるいは ICU のバイナリが非標準的な名前でビルドされている場合に、ライブラリのリンク時に既定の ICU バイナリ名の代わりに *linker-options-for-icu* を使用するには次のようにする。

```
bjam -sICU_LINK="linker-options-for-icu" --toolset=toolset-name install
```

オプション `cxxflags=option` および `linkflags=-option` でコンパイラやリンカに固有のオプションを設定する必要があるかもしれない。

重要

設定の結果はキャッシュされる。異なるコンパイラオプションで再ビルドする場合、bjam のコマンドラインに `-a` を付けるとすべてのターゲットが強制的に再ビルドされる。

ICU がコンパイラのパスに入っておらず、ヘッダ・ライブラリ・バイナリがそれぞれ `path-to-icu/include`、`path-to-icu/lib`、`path-to-icu/bin` にあるのであれば、環境変数 `ICU_PATH` でインストールした ICU のルートディレクトリを指定する必要がある。典型的なのは MSVC でビルドする場合である。例えば ICU を `c:\download\icu` にインストールした場合は、次のようにする。

```
bjam -sICU_PATH=c:\download\icu --toolset=toolset-name install
```

重要

ICU も Boost と同様に C++ライブラリであり、ICU のコピーが Boost のビルドに使用したものと同じ C++コンパイラ（およびバージョン）でビルドされていなければならないということに注意していただきたい。そうでない場合 Boost.Regex は正しく動作しない。

結局のところ、複数のコンパイラのバージョンで異なる ICU ビルド使用してビルド・テストするのであれば、設定の段階で ICU が自動的に検出されるよう各ツールセットに適切なコンパイラ・リンカオプションを設定するよう（ICU バイナリが標準的な名前を使っているのであれば、適切なヘッダとリンカの検索パスを追加するだけでよい）user-config.jam を修正するのが現時点で唯一の方法である。

メイクファイルを使ったビルド

Regex ライブラリは「ただのソースファイル群」であり、ビルドに特に必要なことはない。

<boost のパス>/libs/regex/src*.cpp のファイルをライブラリとしてビルドするか、これらのファイルをあなたのプロジェクトに追加するとよい。既定の Boost ビルドでサポートされていない個々のコンパイラオプションを使う必要がある場合に特に有用である。

以下の 2 つの#define を知っておく必要がある。

- ICU サポートを有効にしてコンパイルする場合は BOOST_HAS_ICU を定義しなければならない。
- Windows で DLL をビルドする場合は BOOST_REGEX_DYN_LINK を定義しなければならない。

重要

Boost.Regex で提供しているメイクファイルは非推奨となったため、次回のリリースで削除する。

導入と概要

正規表現はテキスト処理でよく使われるパターンマッチの形式である。Unix ユーティリティの grep、sed、awk やプログラミング言語 Perl といった正規表現を広範にわたって利用しているツールに馴染みのあるユーザは多い。古くから C++ユーザが正規表現処理を行う方法は POSIX C API に限られていた。Boost.Regex もこれらの API を提供するが、本ライブラリの最適な利用法ではない。例えば Boost.Regex はワイド文字列に対応している。また従来の C ライブラリでは不可能だった（sed や Perl で使用されているような）検索や置換も可能である。

basic_regex は本ライブラリのキーとなるクラスであり、「マシン可読の」正規表現を表す。正規表現パターンは文字列であると同時に、正規表現アルゴリズムに要求される状態マシンでもあるという観点から、`std::basic_string` に非常に近いモデリングになっている。`std::basic_string` と同様に、このクラスを参照するのにほとんど常に使われる typedef が 2 つある。

```
namespace boost {

template <class charT,
          class traits = regex_traits<charT> >
class basic_regex;

typedef basic_regex<char> regex;
typedef basic_regex<wchar_t> wregex;

}
```

このライブラリの使い方を見るために、クレジットカードを処理するアプリケーションを考える。クレジットカードの番号は、通常 16 桁の数字が空白かハイフンで 4 桁ずつのグループに分けられた文字列になっている。クレジットカードの番号をデータベースに格納する前に（顧客にとってはどうでもいいことだろうが）番号が正しい形式になっているか確認したいと思う。あらゆる（10 進）数字にマッチする正規表現として `[0-9]` が使えるが、このような文字の範囲は実際にはロカール依存である。代わりに使用すべきなのは POSIX 標準形式の `[[:digit:]]` か、Boost.Regex および Perl での短縮形である `\d` となる（古いライブラリは C ロカールについてハードコードされていることが多く、結果的に問題となっていなかったことに注意していただきたい）。以上のことから、次の正規表現を使えばクレジットカードの番号形式を検証できる。

```
(\d{4}[- ]){3}\d{4}
```

式中の括弧は部分式のグループ化（および後で参照するためのマーク付け）を行い、`{4}` は「4 回ちょうどの繰り返し」を意味する これは Perl、awk および egrep で使われている拡張正規表現構文の例である。Boost.Regex は sed や grep で使われている古い「基本的な」構文もサポートしているが、基本的な正規表現がすでにあって再利用しようと考えているのでなければ、通常はあまり役に立たない。

では、この式を使ってクレジットカード番号を検証する C++コードを書いてみる。

```
bool validate_card_format(const std::string& s)
{
   static const boost::regex e("(\\d{4}[- ]){3}\\d{4}");
```

```
   return regex_match(s, e);
}
```

式にエスケープが追加されていることに注意していただきたい。エスケープは正規表現エンジンに処理される前に C++コンパイラによって処理されるため、結局、正規表現中のエスケープは C/C++コードでは二重にしなければならない。また、本文書の例はすべてコンパイラが引数依存の名前探索をサポートしているものとしているという点に注意していただきたい。未サポートのコンパイラ（VC6 など）では関数呼び出しの前に boost::を付けなければならない場合がある。

クレジットカードの処理に詳しい人であれば、上の形式が人にとって可読性は高いものの、オンラインのクレジットカードシステムに適した形式になっていないと気づくと思う。こういうシステムでは、間に空白の入らない 16 桁（あるいは 15 桁）の文字列を使う。ここで必要なのは 2 つの形式を簡単に交換する方法であり、こういう場合に検索と置換を使う。Perl や sed といったユーティリティに詳しいのであれば、この部分は読み飛ばしてもらって構わない。ここで必要なのは 2 つの文字列である。1 つは正規表現であり、もう 1 つはマッチしたテキストをどのように置換するか指定する「書式化文字列」である。Boost.Regex でこの検索・置換操作はアルゴリズム regex_replace で行う。今のクレジットカードの例では形式を変換するアルゴリズムを 2 つ記述できる。

```
// いずれの形式にもマッチする正規表現：
const boost::regex e("\\A(\\d{3,4})[- ]?(\\d{4})[- ]?(\\d{4})[- ]?(\\d{4})\\z");
const std::string machine_format("\\1\\2\\3\\4");
const std::string human_format("\\1-\\2-\\3-\\4");

std::string machine_readable_card_number(const std::string s)
{
   return regex_replace(s, e, machine_format, boost::match_default | boost::format_sed);
}

std::string human_readable_card_number(const std::string s)
{
   return regex_replace(s, e, human_format, boost::match_default | boost::format_sed);
}
```

カード番号の 4 つの部分を別々のフィールドに分けるのに、正規表現中でマーク済み部分式を用いた。形式化文字列では、マッチしたテキストを置換するのに sed ライクの構文を使っている。

上の例では正規表現マッチの結果を直接操作することはしなかったが、通常はマッチ結果にはマッチ全体だけでなく部分マッチに関する情報が含まれている。必要な場合はライブラリの match_results クラスのインスタンスを使うとよい。先ほどと同様に、実際に使用する場合は typedef を使うとよい。

```
namespace boost{

typedef match_results<const char*> cmatch;
typedef match_results<const wchar_t*> wcmatch;
typedef match_results<std::string::const_iterator> smatch;
typedef match_results<std::wstring::const_iterator> wsmatch;

}
```

アルゴリズム regex_search および regex_match は、match_results を使ってどこがマッチしたかを返す。2 つのアルゴリズムの違いは regex_match が入力テキスト**全体**に対するマッチを検索するのみであるに対し、regex_search は入力テキスト中のあら

ゆる位置のマッチを検索するということである。

これらのアルゴリズムが、通常の C 文字列の検索に限定されていないことに注意していただきたい。双方向イテレータであれば何でも検索可能であり、ほとんどあらゆる種類のデータにシームレスに対応している。

検索・置換操作については、すでに見た regex_replace に加えて match_results クラスに format メンバがある。このメンバ関数はマッチ結果と書式化文字列を取り、この 2 つをマージして新文字列を生成する。

テキスト中のマッチ結果をすべて走査するイテレータ型が 2 つある。regex_iterator は見つかった match_results オブジェクトを列挙する。一方 regex_token_iterator は文字列の列を列挙する（Perl スタイルの分割操作に似ている）。

テンプレートを使うのが嫌な人には、低水準のテンプレートコードをカプセル化した高水準のラッパクラス RegEx がある。ライブラリの全機能は必要ないという人向けの簡単なインターフェイスとなっており、ナロー文字と「拡張」正規表現構文のみをサポートする。正規表現 C++標準ライブラリの草案には含まれておらず、現在は非推奨である。

POSIX API 関数 regcomp、regexec、regfree および regerror はナロー文字および Unicode 版の両方で利用可能であり、これらの API との互換性が必要な場合のために提供している。

最後に、本ライブラリは GNU および BSD4 の regex パッケージや PCRE、Perl 5 といった正規表現ライブラリの他に、実行時の地域化と完全な POSIX 正規表現構文もサポートしている。これには複数文字の照合要素や等価クラスのような発展的な機能も含まれている。

UnicodeとBoost.Regex

Boost.Regex で Unicode 文字列を使う方法は 2 つある。

wchar_t への依存

プラットフォームの `wchar_t` 型が Unicode 文字列を保持でき、かつプラットフォームの C/C++実行時ライブラリがワイド文字定数（が `std::iswspace` や `std::iswlower` に渡されるなどのケース）を正しく処理できるのであれば、`boost::wregex` を使った Unicode 処理が可能である。しかしながら、このアプローチにはいくつか不便がある。

- 移植性がない。`wchar_t` の幅や実行時ライブラリがワイド文字を Unicode として扱うかどうかについては何の保証もない。ほとんどの Windows コンパイラは保証しているが、多くの Unix システムではそうではない。
- Unicode 固有の文字クラスはサポートされない（`[[:Nd:]]`、`[[:Po:]]`など）。
- ワイド文字シーケンスで符号化された文字列しか検索できない。UTF-8 や UTF-16 でさえも多くのプラットフォームで検索できない。

Unicode 対応の正規表現型の使用

ICU ライブラリがあれば Boost.Regex から利用できるように設定できる。これにより Unicode 固有の文字プロパティや、UTF-8、UTF-16、および UTF-32 で符号化された文字列の検索をサポートする特別な正規表現型（`boost::u32regex`）が提供される。ICU 文字列クラスのサポートを見よ。

マーク済み部分式と捕捉の理解

捕捉とは、正規表現にマッチしたマーク済み部分式に「捕捉された」イテレータ範囲である。マーク済み部分式が複数回マッチした場合は、1 つのマーク済み部分式が複数の捕捉について対応する可能性がある。本文書では捕捉とマーク済み部分式の Boost.Regex での表現とアクセス方法について述べる。

マーク済み部分式

Perl は括弧グループ `()` を含む正規表現について、マーク済み部分式という追加のフィールドを吐き出す。例えば次の正規表現にはマーク済み部分式が 2 つある（それぞれ$1、$2 という）。

```
(\w+)\W+(\w+)
```

また、マッチ全体を $&、最初のマッチより前すべてを $`、マッチより後ろすべてを $' であらわす。よって `"@abc def--"` に対して上の式で検索をかけると次の結果を得る。

部分式	検索されるテキスト
`$``	“@”
`$&`	“abc def”
`$1`	“abc”
`$2`	“def”
`$'`	“--”

Boost.Regex ではこれらはすべて、正規表現マッチアルゴリズム（regex_search、regex_match、regex_iterator）のいずれかを呼び出したときに値が埋められる match_results クラスによりアクセスできる。以下が与えられていたとすると、

```
boost::match_results<IteratorType> m;
```

Perl と Boost.Regex の対応は次のようになる。

Perl	Boost.Regex
$`	`m.prefix()`
$&	`m[0]`
$n	`m[n]`
$'	`m.suffix()`

Boost.Regex では各部分式マッチは sub_match オブジェクトで表現される。これは基本的には部分式がマッチした位置の先頭と終端を指すイテレータの組に過ぎないが、sub_match オブジェクトが `std::basic_string` に類似した振る舞いをするように、演算子がいくつか追加されている。例えば `basic_string` への暗黙の型変換により、文字列との比較、文字列への追加、および出力ストリームへの出力が可能になっている。

マッチしなかった部分式

マッチが見つかったとして、すべてのマーク済み部分式が関与する必要のない場合がある。例えば、

```
(abc)|(def)
```

この式は$1 か$2 のいずれかがマッチする可能性があるが、両方とも同時にマッチすることはない。Boost.Regex では sub_match::matched データメンバにアクセスすることでマッチしたかどうか調べることができる。

捕捉の繰り返し

マーク済み部分式が繰り返されている場合、その部分式は複数回「捕捉される」。しかし通常利用可能なのは最後の捕捉のみである。例えば、

```
(?:(\w+)\W+)+
```

この式は以下にマッチした場合、

```
one fine day
```

$1 には文字列 “day” が格納され、それ以前の捕捉はすべて捨てられる。

しかしながら Boost.Regex の実験的な機能を使用すれば捕捉情報をすべて記憶しておくことが可能である。これにアクセスするには match_results::captures メンバ関数か sub_match::captures メンバ関数を使う。これらの関数は、正規表現マッチの間に記憶した捕捉をすべて含んだコンテナを返す。以下のサンプルプログラムでこの情報の使用方法を説明する。

```
#include</.>
#include<>

(::&,::&)
{
   ::();
   ::;
   ::<<"正規表現：\""<<<<"\"\n";
   ::<<"テキスト：\""<<<<"\"\n";
   (::(,,,::))
   {
      ,;
      ::<<"** マッチが見つかりました **\n 部分式：\n";
      (=0;<.();++)
         ::<<" $"<<<<" = \""<<[]<<"\"\n";
      ::<<" 捕捉：\n";
      (=0;<.();++)
      {
         ::<<" $"<<<<" = {";
         (=0;<.().();++)
         {
            ()
               ::<<", ";
```

```
::<<" ";
            ::<<"\""<<.()[]<<"\"";
         }
         ::<<" }\n";
      }
   }

{
      ::<<"** マッチは見つかりません **\n";
   }
(,*[])
{
   ("((([[:lower:]]+)|([[:upper:]]+))+","aBBcccDDDDDeeeeeeee");
   ("(.*)bar|(.*)bah", "abcbar");
   ("(.*)bar|(.*)bah","abcbah");
   ("^(?:(\\w+)|(?>\\W+))*$",
      "now is the time for all good men to come to the aid of the party");
   0;
}
```

このプログラムの出力は次のようになる。

```
正規表現："((([[:lower:]]+)|([[:upper:]]+))+"
テキスト："aBBcccDDDDDeeeeeeee"
** マッチが見つかりました **
   部分式：
      $0 = "aBBcccDDDDDeeeeeeee"
      $1 = "eeeeeeee"
      $2 = "eeeeeeee"
      $3 = "DDDDD"
   捕捉：
      $0 = { "aBBcccDDDDDeeeeeeee" }
      $1 = { "a", "BB", "ccc", "DDDDD", "eeeeeeee" }
      $2 = { "a", "ccc", "eeeeeeee" }
      $3 = { "BB", "DDDDD" }
正規表現："(.*)bar|(.*)bah"
テキスト："abcbar"
** マッチが見つかりました **
   部分式：
      $0 = "abcbar"
      $1 = "abc"
      $2 = ""
   捕捉：
      $0 = { "abcbar" }
      $1 = { "abc" }
      $2 = { }
正規表現："(.*)bar|(.*)bah"
テキスト："abcbah"
** マッチが見つかりました **
   部分式：
      $0 = "abcbah"
      $1 = ""
      $2 = "abc"
   捕捉：
      $0 = { "abcbah" }
```

```
      $1 = { }
      $2 = { "abc" }
正規表現："^(?:(\w+)|(?>\W+))*$"
テキスト："now is the time for all good men to come to the aid of the party"
** マッチが見つかりました **
   部分式：
      $0 = "now is the time for all good men to come to the aid of the party"
      $1 = "party"
   捕捉：
      $0 = { "now is the time for all good men to come to the aid of the party" }
      $1 = { "now", "is", "the", "time", "for", "all", "good", "men", "to",
         "come", "to", "the", "aid", "of", "the", "party" }
```

残念ながらこの機能を有効にすると（実際に使用しない場合でも）効率に影響が出る上、使用した場合はさらに効率が悪化するため、以下の 2 つを行わないと使用できないようになっている。

- ライブラリのソースコードをインクルードするすべての翻訳単位で BOOST_REGEX_MATCH_EXTRA を定義する（boost/regex/user.hpp にシンボルの定義部分があるので、このコメントを解除するのが最もよい）。
- 実際に捕捉情報が必要な個々のアルゴリズムで match_extra フラグを渡す（regex_search、regex_match、regex_iterator）。

部分マッチ

アルゴリズム regex_match 、regex_search 、regex_grep およびイテレータ regex_iterator で使用可能な match_flag_type に match_partial がある。このフラグを使うと完全マッチだけでなく部分マッチも検索される。部分マッチは入力テキストの終端 1 文字以上にマッチしたが、正規表現全体にはマッチしなかった（が、さらに入力が追加されれば全体にマッチする可能性のある）場合である。部分マッチを使用する典型的な場合として、入力データの検証（キーボードから文字が入力されるたびにチェックする場合）や、テキスト検索においてテキストがメモリ（あるいはメモリマップドファイル）に読み込めないほど非常に長いか、（ソケットなどから読み込むため）長さが不確定な場合がある。部分マッチと完全マッチの違いを以下の表に示す（変数 *M* は match_results のインスタンスであり、regex_match、regex_search、regex_grep のいずれかの結果が入っているとする）。

	結果	M[0].matched	M[0].first	M[0].second
マッチしなかった場合	偽	未定義	未定義	未定義
部分マッチ	真	偽	部分マッチの先頭	部分マッチの終端（テキストの終端）
完全マッチ	真	真	完全マッチの先頭	完全マッチの終端

部分マッチはいささか不完全な振る舞いをする場合があることに注意していただきたい。

- `.*abc` のようなパターンは常に部分マッチを生成する。この問題を軽減して使用するには、正規表現を注意深く構築するか、match_not_dot_newline のようなフラグを設定して `.*`といったパターンが前の行境界にマッチしないようにする。
- 現時点では Boost.Regex は完全マッチに対して最左マッチを採用しているため、`"ab"` に対して `abc|b` でマッチをかけると `"b"` に対する完全マッチではなく `"ab"` に対する部分マッチが得られる。

次の例は、テキストが正しいクレジットカード番号となり得るかを、調べる。ユーザが打鍵して入力された文字が文字列に追加されるたびに、文字列を is_possible_card_number に渡すという使い方を想定している。この手続きが真を返す場合、テキストは正しいクレジットカード番号であり、ユーザインターフェイスの OK ボタンを有効にする。偽を返す場合、テキストはまだ正しいカード番号になっていないが、さらに入力があれば正しい番号となるため、ユーザインターフェイスの OK ボタンを無効にする。最後に手続きが例外を投げる場合は、入力が正しい番号となる可能性が無いため、入力テキストを破棄して適切なエラーをユーザに表示しなければならない。

```
#include <string>
#include <iostream>
#include <boost/regex.hpp>

boost::regex e("(\\d{3,4})[- ]?(\\d{4})[- ]?(\\d{4})[- ]?(\\d{4})");

bool is_possible_card_number(const std::string& input)
{
   //
   // 部分マッチに対しては偽、完全マッチに対しては真を返す。
   // マッチの可能性がない場合は例外を投げる…
   boost::match_results<std::string::const_iterator> what;
   if(0 == boost::regex_match(input, what, e, boost::match_default | boost::match_partial))
```

```
   {
      // 入力が正しい形式となる可能性はなくなったので拒絶する：
      throw std::runtime_error(
         "不正なデータが入力されました - 追加の入力があっても正しい番号となる可能性はありません");
   }
   // OK、今のところはよろしい。だが、入力はこれで終わりだろうか？
   if(what[0].matched)
   {
      // 素晴らしい。正しい結果が得られた：
      return true;
   }
   // この時点では部分的にマッチしただけ…
   return false;
}
```

次の例では、入力テキストは長さが未知であるストリームから取得する。この例は単純にストリーム中で見つかった HTML タグの数を数える。テキストはバッファに読み込まれ、1 度に一部分だけを検索する。部分マッチが見つかった場合、さらにその部分マッチを次のテキスト群の先頭として検索を行う。

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <boost/regex.hpp>

// HTML タグにマッチする：
boost::regex e("<[^>]*>");
// タグの数：
unsigned int tags = 0;

void search(std::istream& is)
{
   // 検索するバッファ：
   char buf[4096];
   // 部分マッチの先頭位置を保存：
   const char* next_pos = buf + sizeof(buf);
   // 入力がまだあるかを示すフラグ：
   bool have_more = true;

   while(have_more)
   {
      // 前回の試行から何文字コピーするか：
      unsigned leftover = (buf + sizeof(buf)) - next_pos;
      // および、ストリームから何文字読み込むか：
      unsigned size = next_pos - buf;
      // 前回残っていた部分をバッファの先頭にコピー：
      std::memmove(buf, next_pos, leftover);
      // 残りをストリームからの入力で埋める：
      is.read(buf + leftover, size);
      unsigned read = is.gcount();
      // テキストをすべて走査したかチェック：
      have_more = read == size;
      // next_pos をリセット：
```

```
      next_pos = buf + sizeof(buf);
      // 走査を行う：
      boost::cregex_iterator a(
         buf,
         buf + read + leftover,
         e,
         boost::match_default | boost::match_partial);
      boost::cregex_iterator b;

      while(a != b)
      {
         if((*a)[0].matched == false)
         {
            // 部分マッチ。位置を保存しループを脱出：
            next_pos = (*a)[0].first;
            break;
         }
         else
         {
            // 完全マッチ：
            ++tags;
         }

         // 次のマッチへ移動：
         ++a;
      }
   }
}
```

正規表現の構文

本節では本ライブラリで使用可能な正規表現構文について述べる。本文書はプログラミングガイドであり、あなたのプログラムのユーザが触れる実際の構文は正規表現をコンパイルするときに用いるフラグによる。

正規表現オブジェクトの構築方法により、主に 3 つの構文が用意されている。

- Perl（既定の動作）。
- POSIX 拡張（egrep および awk の変種も含む）。
- POSIX 基本（grep および emacs の変種も含む）。

すべての文字を直値（リテラル）として扱う正規表現も構築可能である（実際のところ「構文」とはいえないが）。

Perl の正規表現構文

概要

Perl の正規表現構文は、プログラミング言語 Perl で使われているものに基づいている。Perl の正規表現は Boost.Regex の既定の動作であり、`basic_regex` のコンストラクタにフラグ `perl` を渡すことでも利用できる。例えば以下のとおり。

```
// e1 は大文字小文字を区別する Perl の正規表現：
// Perl は既定のオプションであり、明示的に構文を指定する必要はない：
boost::regex e1(my_expression);
// e2 は大文字小文字を区別しない Perl の正規表現：
boost::regex e2(my_expression, boost::regex::perl|boost::regex::icase);
```

Perl の正規表現構文

Perl の正規表現では、以下の特別なものを除くあらゆる文字が文字そのものにマッチする。

```
.[{()\*+?|^$
```

ワイルドカード

文字集合外部の `.` 1 文字は、以下以外のあらゆる文字 1 文字にマッチする。

- NULL 文字（マッチアルゴリズムにフラグ `match_not_dot_null` を渡した場合）。
- 改行文字（マッチアルゴリズムにフラグ `match_not_dot_newline` を渡した場合）。

アンカー

^は行頭にマッチする。

$は行末にマッチする。

マーク済み部分式

開始が(で終了が)の節は部分式として機能する。マッチした部分式はすべてマッチアルゴリズムにより個別のフィールドに分けられる。マーク済み部分式は繰り返しと後方参照により参照が可能である。

マークなしのグループ化

マーク済み部分式は正規表現を字句的なグループに分けるのに役立つが、結果的に余分なフィールドを生成するという副作用がある。マーク済み部分式を生成することなく正規表現を字句的なグループに分ける別の手段として、(?:と)を使う方法がある。例えば(?:ab)+はabの繰り返しを表し、別個の部分式を生成しない。

繰り返し

あらゆるアトム（文字、部分式、文字クラス）は *、+、? および {} 演算子による繰り返しが可能である。

*演算子は直前のアトムの 0 回以上の繰り返しにマッチする。例えば正規表現 a*b は以下のいずれにもマッチする。

```
b
ab
aaaaaaaab
```

+演算子は直前のアトムの 1 回以上の繰り返しにマッチする。例えば正規表現 a+b は以下のいずれにもマッチする。

```
ab
aaaaaaaab
```

しかし次にはマッチしない。

```
b
```

?演算子は直前のアトムの 0 回あるいは 1 回の出現にマッチする。例えば正規表現 ca?b は以下のいずれにもマッチする。

```
cb
cab
```

しかし次にはマッチしない。

```
caab
```

アトムの繰り返しは回数境界指定の繰り返しによっても可能である。

a{n}は "a" のちょうど n 回の繰り返しにマッチする。

a{n,}は "a" の n 回以上の繰り返しにマッチする。

a{n,m}は "a" の n 回以上 m 回以下の繰り返しにマッチする。

例えば

```
^a{2,3}$
```

は、次のいずれにもマッチするが、

```
aa
aaa
```

次のいずれにもマッチしない。

```
a
aaaa
```

文字{および}は、繰り返し以外の場面では通常のリテラルとして扱うことに注意していただきたい。これは Perl 5.x と同じ振る舞いである。例えば式 ab{1、ab1}および a{b}c の波括弧はリテラルとして扱い、**エラーは発生しない**。

直前の構造が繰り返し不能な場合に繰り返し演算子を使うとエラーになる。例えば次は

```
a(*)
```

*演算子を適用可能なものがないためエラーとなる。

貪欲でない繰り返し

通常の繰り返し演算子は「貪欲」である。貪欲とは、可能な限り長い入力にマッチするという意味である。マッチを生成する中で最も短い入力に一致する貪欲でないバージョンがある。

*?は直前のアトムの 0 回以上の繰り返しにマッチする最短バージョンである。

+?は直前のアトムの 1 回以上の繰り返しにマッチする最短バージョンである。

??は直前のアトムの 0 回か 1 回の出現にマッチする最短バージョンである。

{n,}?は直前のアトムの n 回以上の繰り返しにマッチする最短バージョンである。

{n,m}?は直前のアトムの n 回以上 m 回以下の繰り返しにマッチする最短バージョンである。

強欲な繰り返し

既定では、繰り返しパターンがマッチに失敗すると正規表現エンジンはマッチが見つかるまでバックトラッキングを行う。しかしながらこの動作が不都合な場合があるため、「強欲な」繰り返しというものがある。可能な限り長い文字列にマッチするが、式の残りの部分がマッチに失敗してもバックトラックを行わない。

*+は直前のアトムの 0 回以上の繰り返しにマッチし、バックトラックを行わない。

++は直前のアトムの 1 回以上の繰り返しにマッチし、バックトラックを行わない。

?+は直前のアトムの 0 回か 1 回の出現にマッチし、バックトラックを行わない。

{n,}+は直前のアトムの n 回以上の繰り返しにマッチし、バックトラックを行わない。

{n,m}+は直前のアトムの n 回以上 m 回以下の繰り返しにマッチし、バックトラックを行わない。

後方参照

エスケープ文字の直後に数字 *n* があると、部分式 *n* にマッチしたものと同じ文字列にマッチする。*n* は 0 から 9 の範囲である。例えば次の正規表現は、

```
^(a*).*\1$
```

次の文字列にマッチする。

```
aaabbaaa
```

しかし、次の文字列にはマッチしない。

```
aaabba
```

\gエスケープを使用しても同じ効果が得られる。例えば、

エスケープ	意味
\g1	1 番目の部分式にマッチ。
\g{1}	1 番目の部分式にマッチ。この形式を使うと\g{1}2のような式や、\g{1234}といった添字が 9 より大きい式を安全に解析できる。
\g-1	最後の部分式にマッチ。
\g{-2}	最後から 2 番目の部分式にマッチ。
\g{one}	“one” という名前の部分式にマッチ。

最後に、\kエスケープで名前付き部分式を参照できる。例えば\k<two>は “two” という名前の部分式にマッチする。

選択

|演算子は引数のいずれかにマッチする。よって、例えばabc|defは“abc”か“def”のいずれかにマッチする。

括弧を使用すると選択をグループ化できる。例えばab(d|ef)は“abd”か“abef”のいずれかにマッチする。

空の選択というのは許されないが、本当に必要な場合はプレースホルダーとして(?:)を使用する。例えば、

- |abcは有効な式ではない。
- しかし、(?:)|abcは有効な式であり、実現しようとしていることは同じである。
- (?:abc)??も全く同じ意味である。

文字集合

文字集合は`[`で始まり`]`で終わる括弧式であり、文字の集合を定義する。集合に含まれるいずれかの 1 文字にマッチする。

文字集合に含められる要素は以下の組み合わせである。

単一の文字

例えば`[abc]`は `"a"`、`"b"`、`"c"` のいずれか 1 文字にマッチする。

文字範囲

例えば`[a-c]`は ‘a’ から ‘c’ までの範囲の 1 文字にマッチする。Perl の正規表現の既定では、文字 *x* が *y* から *z* の範囲であるとは、文字のコードポイントが範囲の端点を含んだコードポイント内にある場合をいう。ただし、正規表現の構築時に `collate` フラグ設定するとこの範囲はロカール依存となる。

否定

括弧式が文字`^`で始まっている場合は、正規表現に含まれる文字の補集合となる。例えば`[^a-c]`は範囲 `a-c` を除くあらゆる文字にマッチする。

文字クラス

`[[:name:]]`のような形式の正規表現は名前付き文字クラス「name」にマッチする。例えば`[[:lower:]]`はあらゆる小文字にマッチする。文字クラス名を見よ。

照合要素

`[[.col.]]`のような形式の式は照合要素 *col* にマッチする。照合要素とは、単一の照合単位として扱われる文字か文字シーケンスである。照合要素は範囲の端点としても使用できる。例えば`[[.ae.]-c]`は文字シーケンス “ae” のみならず、範囲 “ae”-c のいずれか 1 文字にもマッチする。後者において “ae” は現在のロカールにおける単一の照合要素として扱われる。

この拡張として、照合要素をシンボル名で指定する方法もある。例えば、

```
[[.NUL.]]
```

は文字`"\0"`にマッチする。

等価クラス

`[[=col=]]`のような形式の正規表現は、第 1 位のソートキーが照合要素 *col* と同じ文字および照合要素にマッチする。照合要素名 *col* はシンボル名でもよい。第 1 位のソートキーでは大文字小文字の違い、アクセント記号の有無、ロカール固有のテーラリング[3]は無視される。よって`[[=a=]]`は a、À、Á、Â、Ã、Ä、Å、A、à、á、â、ã、ä および å のいずれにもマッチする。残念ながらこの機能の実装はプラットフォームの照合と地域化のサポートに依存し、すべてのプラットフォームで移植性の高い動作は期待できず、単一の

3 訳注 テーラリング（tailoring）は汎用的な処理に対して、特定の事情に即した結果を得るために追加規則を用いて処理をカスタマイズすることを意味します。文字処理の分野では特に各ロカールに対応するためのテーラリングが多数存在します。

プラットフォームにおいてもすべてのロカールで動作するとは限らない。

エスケープ付き文字

1 文字にマッチするエスケープシーケンスおよび文字クラスが、文字クラスの定義で使用可能である。例えば`[\[\]]`は"["、"]"のいずれかにマッチする。また`[\W\d]`は「数字」か、「単語」**でない** 1 文字にマッチする。

結合

以上の要素はすべて 1 つの文字集合宣言内で結合可能である。例: `[[:digit:]a-c[.NUL.]]`

エスケープ

直前にエスケープの付いた特殊文字は、すべてその文字自身にマッチする。

以下のエスケープシーケンスは、すべて 1 文字の別名である。

エスケープ	文字
`\a`	`\a`
`\e`	`0x1B`
`\f`	`\f`
`\n`	`\n`
`\r`	`\r`
`\t`	`\t`
`\v`	`\v`
`\b`	`\b`（文字クラス宣言内のみ）
`\cX`	ASCII エスケープシーケンス。コードポイントが X % 32 の文字
`\xdd`	16 進エスケープシーケンス。コードポイントが 0xdd の文字にマッチする。
`\x{dddd}`	16 進エスケープシーケンス。コードポイントが 0xdddd の文字にマッチする。
`\0ddd`	8 進エスケープシーケンス。コードポイントが 0ddd の文字にマッチする。
`\N{name}`	シンボル名 *name* の文字にマッチする。例えば`\N{newline}`は文字 "\n" にマッチする。

「単一文字」文字クラス

x が文字クラス名である場合、エスケープ文字 *x* はその文字クラスに属するあらゆる文字にマッチし、エスケープ文字 *X* はその文字クラスに属さないあらゆる文字にマッチする。

既定でサポートされているものは以下のとおりである。

エスケープシーケンス	等価な文字クラス
`\d`	`[[:digit:]]`
`\l`	`[[:lower:]]`
`\s`	`[[:space:]]`
`\u`	`[[:upper:]]`
`\w`	`[[:word:]]`

\h	水平空白
\v	垂直空白
\D	[^[:digit:]]
\L	[^[:lower]]
\S	[^[:space:]]
\U	[^[:upper:]]
\W	[^[:word:]]
\H	水平空白以外
\V	垂直空白以外

文字プロパティ

次の表の文字プロパティ名はすべて文字クラスで使用する名前と等価である。

形式	説明	等価な文字集合の形式
\pX	プロパティ*X*をもつあらゆる文字にマッチする。	[[:X:]]
\p{Name}	プロパティ*Name*をもつあらゆる文字にマッチする。	[[:Name:]]
\PX	プロパティ*X*をもたないあらゆる文字にマッチする。	[^[:X:]]
\P{Name}	プロパティ*Name*をもたないあらゆる文字にマッチする。	[^[:Name:]]

例えば\pdは\p{digit}と同様、あらゆる「数字」（“digit”）にマッチする。

単語境界

次のエスケープシーケンスは単語の境界にマッチする。

\<は単語の先頭にマッチする。

\>は単語の終端にマッチする。

\bは単語境界（単語の先頭か終端）にマッチする。

\Bは単語境界以外にマッチする。

バッファ境界

以下はバッファ境界にのみマッチする。この場合の「バッファ」とは、マッチ対象の入力テキスト全体である（^および$はテキスト中の改行にもマッチすることに注意していただきたい）。

\`はバッファの先頭にのみマッチする。

\'はバッファの終端にのみマッチする。

\Aはバッファの先頭にのみマッチする（\`と同じ）。

\zはバッファの終端にのみマッチする（\'と同じ）。

\Zはバッファ終端における省略可能な改行シーケンスのゼロ幅表明にマッチする。正規表現(?=\v*\z)と等価である。(?=\n?\z)のような動作をするPerlとは微妙に異なることに注意していただきたい。

継続エスケープ（Continuation Escape）

シーケンス`\G`は最後にマッチが見つかった位置、あるいは前回のマッチが存在しない場合はマッチ対象テキストの先頭にのみマッチする。各マッチが 1 つ前のマッチの終端から始まっているようなマッチをテキスト中から列挙する場合に、このシーケンスは有効である。

クォーティングエスケープ（Quoting Escape）

エスケープシーケンス`\Q`は「クォートされたシーケンス」の開始を表す。以降、正規表現の終端か`\E`までの文字はすべて直値として扱われる。例えば、正規表現`\Q\*+\Ea+`は以下のいずれかにマッチする。

```
\*+a
\*+aaa
```

Unicode エスケープ

`\C`は単一のコードポイントにマッチする。Boost.Regex では`.`演算子とまったく同じ意味である。`\X`は結合文字シーケンス（非結合文字に 0 以上の結合文字シーケンスが続く）にマッチする。

行末へのマッチ

エスケープシーケンス`\R`はあらゆる改行文字シーケンスにマッチする。つまり、式`(?>\x0D\x0A?|[\x0A-\x0C\x85\x{2028}\x{2029}])`と等価である。

テキストの除外

`\K`は$0 の開始位置を現在のテキスト位置にリセットする。言い換えると`\K`より前にあるものはすべて差し引かれ、正規表現マッチの一部とならない。$`も同様に更新される。

例えば`foo\Kbar`をテキスト`"foobar"`にマッチさせると、$0 に対して`"bar"`、$`に対して`"foo"`というマッチ結果が返る。これは可変幅の後方先読みを再現するのに使用する。

その他のエスケープ

その他のエスケープシーケンスは、エスケープ対象の文字そのものにマッチする。例えば`\@`は直値`"@"`にマッチする。

Perl の拡張パターン

正規表現構文の Perl 固有の拡張はすべて`(?`で始まる。

名前付き部分式

以下のようにして部分式を作成する。

```
(?<NAME>expression)
```

これで *NAME* という名前で参照可能になる。あるいは以下の`'NAME'`のように区切り文字を使う方法もある。

```
(?'NAME'expression)
```

これらの名前付き部分式は後方参照内で`\g{NAME}`か`\k<NAME>`で参照する。検索・置換操作で使用する Perl 形式の文字列、および`match_results`メンバ関数では名前で参照する。

注釈

`(?# ... )`は注釈（コメント）として扱われ、内容は無視される。

修飾子

`(?imsx-imsx ... )`は、パターン中でどの Perl 修飾子を有効にするかを設定する。効果はブロックの先頭から閉じ括弧`)`までである。`-`より前にある文字が Perl 修飾子を有効にし、後にある文字が無効にする。

`(?imsx-imsx:pattern)`は、指定した修飾子をパターンのみに適用する。

マークなしのグループ

`(?:pattern)`は、パターンを字句的にグループ化する。部分式の生成はない。

選択分岐ごとの部分式番号のリセット（Branch reset）

`(?|pattern)`は、*pattern* 内において`|`が現れるごとに部分式の番号をリセットする。

この構造の後ろの部分式番号は、部分式の数が最大である選択分岐により決定する。この構造は、複数の選択マッチから 1 つを単一の部分式添字で捕捉したい場合に有効である。

以下に例を示す。式の下にあるのが各部分式の添字である。

```
# before  ---------------branch-reset----------- after
/ ( a )  (?| x ( y ) z | (p (q) r) | (t) u (v) ) ( z ) /x
# 1            2         2  3        2     3     4
```

先読み

`(?=pattern)`はパターンがマッチした場合に限り、現在位置を進めない。

`(?!pattern)`はパターンがマッチしなかった場合に限り、現在位置を進めない。

先読みを使用する典型的な理由は、2 つの正規表現の論理和作成である。例えばパスワードが大文字、小文字、区切り記号を含み、6 文字以上でなければならないとすると、次の正規表現でパスワードを検証できる。

```
(?=.*[[:lower:]])(?=.*[[:upper:]])(?=.*[[:punct:]]).{6,}
```

後読み

`(?<=pattern)`は、現在位置の直前の文字列がパターンにマッチ可能な場合に限り、現在位置を進めない（パターンは固定長でなければならない）。

`(?<!pattern)`は、現在位置の直前の文字列がパターンにマッチ不能な場合に限り、現在位置を進めない（パターンは固定長でなければならない）。

独立部分式

`(?>pattern)`とすると、*pattern* は周囲のパターンとは独立してマッチし、正規表現は *pattern* にはバックトラックしない。独立部分式を使用する典型的な理由は効率の向上である。可能な限り最良のマッチのみが考慮されるため、独立部分式が正規表現全体のマッチを妨害する場合はマッチは 1 つも見つからない。[4]

再帰式

`(?N)` `(?-N)` `(?+N)` `(?R)` `(?0)`

`(?R)`および`(?0)`はパターン全体の先頭に再帰する。

`(?N)`は *N* 番目の部分式を再帰的に実行する。例えば`(?2)`は 2 番目の部分式へ再帰する。

`(?-N)`および`(?+N)`は相対的な再帰である。例えば`(?-1)`は最後の部分式へ、`(?+1)`は次の部分式へ再帰する。

条件式

`(?(condition)yes-pattern|no-pattern)`は、*condition* が真であれば *yes-pattern*、それ以外の場合は *no-pattern* のマッチを行う。

`(?(condition)yes-pattern)`は、*condition* が真であれば *yes-pattern* のマッチを行い、それ以外の場合は空文字列にマッチする。

condition は前方先読み表明、マーク済み部分式の添字（対応する部分式がマッチしていれば条件が真）、あるいは再帰式の添字（指定した再帰式内を直接実行している場合に条件が真）のいずれかである。

考えられる条件式を挙げる。

- `(?(?=assert)yes-pattern|no-pattern)`は、前方先読み表明がマッチした場合に *yes-pattern* を、それ以外の場合に *no-pattern* を実行する。
- `(?(?!assert)yes-pattern|no-pattern)`は、前方先読み表明がマッチしなかった場合に *yes-pattern* を、それ以外の場合に *no-pattern* を実行する。
- `(?(N)yes-pattern|no-pattern)`は、*N* 番目の部分式がマッチした場合に *yes-pattern* を、それ以外の場合に *no-pattern* を実行する。
- `(?(<name>)yes-pattern|no-pattern)`は、名前付き部分式 *name* がマッチした場合に *yes-pattern* を、それ以外の場合に *no-pattern* を実行する。
- `(?('name')yes-pattern|no-pattern)`は、名前付き部分式 *name* がマッチした場合に *yes-pattern* を、それ以外の場合

4　訳注　独立部分式の知識がある方でなければ意味不明だと思います。Perl のチュートリアル等を参照されることをお勧めします。

に *no-pattern* を実行する。

- `(?(R)yes-pattern|no-pattern)` は、再帰式内を実行中である場合に *yes-pattern* を、それ以外の場合に *no-pattern* を実行する。
- `(?(RN)yes-pattern|no-pattern)` は、*N* 番目の部分式への再帰内を実行中である場合に *yes-pattern* を、それ以外の場合に *no-pattern* を実行する。
- `(?(R&name)yes-pattern|no-pattern)` は、名前付き部分式 *name* への再帰内を実行中である場合に *yes-pattern* を、それ以外の場合に *no-pattern* を実行する。
- `(?(DEFINE)never-executed-pattern)` は絶対に実行されず、どこにもマッチしないコードブロックを定義する。通常、パターン内の別の場所から参照する 1 つ以上の名前付き式を定義するのに使用する。

演算子の優先順位

演算子の優先順位は以下のとおりである。

1. 照合関係の括弧記号 `[==]` `[::]` `[..]`
2. エスケープ `\`
3. 文字集合（括弧式） `[]`
4. グループ `()`
5. 単一文字の繰り返し `*` `+` `?` `{m,n}`
6. 結合
7. アンカー `^$`
8. 選択 `|`

マッチするもの

正規表現を有向グラフ（あるいは閉路グラフ）とみなすと、入力テキストに対する最良マッチとは、グラフに対して深さ優先検索を行って最初に見つかるマッチである。

これは言い換えると次のようになる。最良マッチとは各要素が以下のようにマッチする最左マッチである。

構造	マッチするもの
`AtomA AtomB`	*AtomB* に対するマッチが直後に続く *AtomA* に対する最良マッチを検索する。
`ExpressionA | ExpressionB`	*Expression1* がマッチ可能であればそのマッチを返す。それ以外の場合は *Expression2* を試行する。
`S{N}`	*S* のちょうど *N* 回の繰り返しにマッチする。
`S{N,M}`	*S* の、*N* 回以上 *M* 回以下の可能な限り長い繰り返しにマッチする。
`S{N,M}?`	*S* の、*N* 回以上 *M* 回以下の可能な限り短い繰り返しにマッチする。
`S?`、`S*`、`S+`	それぞれ `S{0,1}`、`S{0,UINT_MAX}`、`S{1,UINT_MAX}` と同じ。

`S??`、`S*?`、`S+?`	それぞれ `S{0,1}?`、`S{0,UINT_MAX}?`、`S{1,UINT_MAX}?`と同じ。
`(?>S)`	*S* の最良マッチにマッチするのみ。
`(?=S)`、`(?<=S)`	*S* の最良マッチにのみマッチする（これが分かるのは、*S* 中に捕捉を行う括弧がある場合のみ）。
`(?!S)`、`(?<!S)`	*S* に対するマッチが存在するかどうか考慮するのみ。
`(?(condition)yes-pattern|no-pattern)`	条件が真であれば *yes-pattern* のみが考慮される。それ以外の場合は *no-pattern* のみが考慮される。

バリエーション

`normal`、`ECMAScript`、`JavaScript` および `JScript` の各オプションはすべて `perl` の別名である。

オプション

正規表現構築時に `perl` オプションとともに指定可能なフラグが多数ある。特に `collate`、`no_subs`、`icase` オプションが大文字小文字の区別やロカール依存の動作を変更するのに対し、`newline_alt` オプションは構文を変更するという点に注意していただきたい。

パターン修飾子

`(?smix-smix)`を正規表現の前につけるか、正規表現コンパイル時フラグ `no_mod_m`、`mod_x`、`mod_s`、`no_mod_s` を使用することで Perl の `smix` 修飾子を適用することができる。

参考

Perl 5.8。[5]

POSIX 拡張正規表現構文

概要

POSIX 拡張正規表現構文は POSIX C 正規表現 API によりサポートされ、`egrep` および `awk` ユーティリティがその変種を使用している。Boost.Regex で POSIX 拡張正規表現を使用するには、コンストラクタにフラグ `extended` を渡す。例えば、

```
// e1 は大文字小文字を区別する POSIX 拡張正規表現：
boost::regex e1(my_expression, boost::regex::extended);
// e2 は大文字小文字を区別しない POSIX 拡張正規表現：
boost::regex e2(my_expression, boost::regex::extended|boost::regex::icase);
```

5 訳注 リンク先はバージョン 5 系列の最新版になっています。現在の Boost.Regex には Perl 5.9 以降の機能が追加されているので、確認しておくといいです。

POSIX 拡張構文

POSIX 拡張正規表現では、以下の特別なものを除くあらゆる文字が文字そのものにマッチする。

```
.[{()\*+?|^$
```

ワイルドカード

文字集合外部の`.`1 文字は、以下以外のあらゆる文字 1 文字にマッチする。

- NULL 文字（マッチアルゴリズムにフラグ `match_not_dot_null` を渡した場合）。
- 改行文字（マッチアルゴリズムにフラグ `match_not_dot_newline` を渡した場合）。

アンカー

`^`は、正規表現の先頭、あるいは部分式の先頭で使用した場合に行頭にマッチする。

`$`は、正規表現の終端、あるいは部分式の終端で使用した場合に行末にマッチする。

マーク済み部分式

開始が`(`で終了が`)`の節は部分式として機能する。マッチした部分式はすべてマッチアルゴリズムにより個別のフィールドに分けられる。マーク済み部分式は繰り返しと後方参照により参照が可能である。

繰り返し

あらゆるアトム（文字、部分式、文字クラス）は `*`、`+`、`?` および `{}` 演算子による繰り返しが可能である。

`*`演算子は直前のアトムの **0 回以上の繰り返し**にマッチする。例えば正規表現 `a*b` は以下のいずれにもマッチする。

```
b
ab
aaaaaaaab
```

`+`演算子は直前のアトムの **1 回以上の繰り返し**にマッチする。例えば正規表現 `a+b` は以下のいずれにもマッチする。

```
ab
aaaaaaaab
```

しかし次にはマッチしない。

```
b
```

`?`演算子は直前のアトムの **0 回あるいは 1 回の出現**にマッチする。例えば正規表現 `ca?b` は以下のいずれにもマッチする。

```
cb
cab
```

しかし次にはマッチしない。

```
caab
```

アトムの繰り返しは回数境界指定の繰り返しによっても可能である。

`a{n}`は `"a"` の**ちょうど n 回の繰り返し**にマッチする。

`a{n,}`は `"a"` の **n 回以上の繰り返し**にマッチする。

`a{n,m}`は `"a"` の **n 回以上 m 回以下の繰り返し**にマッチする。

例えば

```
^a{2,3}$
```

は、次のいずれにもマッチするが、

```
aa
aaa
```

次のいずれにもマッチしない。

```
a
aaaa
```

直前の構造が繰り返し不能な場合に繰り返し演算子を使うとエラーになる。例えば次は

```
a(*)
```

`*`演算子を適用可能なものがないためエラーとなる。

後方参照

エスケープ文字の直後に数字 *n* があると、部分式 *n* にマッチしたものと同じ文字列にマッチする。*n* は 0 から 9 の範囲である。例えば次の正規表現は、

```
^(a*).*\1$
```

次の文字列にマッチする。

```
aaabbaaa
```

しかし、次の文字列にはマッチしない。

```
aaabba
```

注意

POSIX 標準は「拡張」正規表現の後方参照をサポートしない。これは標準に対する互換拡張である。

選択

`|`演算子は引数のいずれかにマッチする。よって、例えば`abc|def`は`"abc"`か`"def"`のいずれかにマッチする。

括弧を使用すると選択をグループ化できる。例えば`ab(d|ef)`は`"abd"`か`"abef"`のいずれかにマッチする。

文字集合

文字集合は`[`で始まり`]`で終わる括弧式であり、文字の集合を定義する。集合に含まれるいずれかの 1 文字にマッチする。

文字集合に含められる要素は以下の組み合わせである。

単一の文字

例えば`[abc]`は`"a"`、`"b"`、`"c"`のいずれか 1 文字にマッチする。

文字範囲

例えば`[a-c]`は'a'から'c'までの範囲の 1 文字にマッチする。POSIX 拡張正規表現の既定では、文字 *x* が *y* から *z* の範囲であるとは、文字の照合順がその範囲内にある場合をいう。結果はロカールの影響を受ける。この動作は`collate`オプションフラグを設定しないことで抑止でき、文字が特定の範囲内にあるかどうかは文字のコードポイントのみで決定する。

否定

括弧式が文字`^`で始まっている場合は、正規表現に含まれる文字の補集合となる。例えば`[^a-c]`は範囲`a-c`を除くあらゆる文字にマッチする。

文字クラス

`[[:name:]]`のような形式の正規表現は名前付き文字クラス「name」にマッチする。例えば`[[:lower:]]`はあらゆる小文字にマッチする。文字クラス名を見よ。

照合要素

`[[.col.]]`のような形式の式は照合要素 *col* にマッチする。照合要素とは、単一の照合単位として扱われる文字か文字シーケンスである。照合要素は範囲の端点としても使用できる。例えば`[[.ae.]-c]`は文字シーケンス “ae” に加えて、現在の範囲 “ae”-c の文字のいずれかにマッチする。 後者において “ae” は現在のロカールにおける単一の照合要素として扱われる。

照合要素は（通常、文字集合内で使用できない）エスケープの代わりとして使用できる。例えば`[[.^.]abc]`は ‘abc^’ のいずれかの 1 文字にマッチする。

この拡張として、照合要素をシンボル名で指定する方法もある。例えば、

```
[[.NUL.]]
```

は NUL 文字にマッチする。

等価クラス

`[[=col=]]`のような形式の正規表現は、第 1 位のソートキーが照合要素 *col* と同じ文字および照合要素にマッチする。照合要素名 *col* はシンボル名でもよい。第 1 位のソートキーでは大文字小文字の違い、アクセント記号の有無、ロカール固有のテーラリング（tailoring）は無視される。よって`[[=a=]]`は a、À、Á、Â、Ã、Ä、Å、A、à、á、â、ã、ä および å のいずれにもマッチする。残念ながらこの機能の実装はプラットフォームの照合と地域化のサポートに依存し、すべてのプラットフォームで移植性の高い動作は期待できず、単一のプラットフォームにおいてもすべてのロカールで動作するとは限らない。

結合

以上の要素はすべて 1 つの文字集合宣言内で結合可能である。例: `[[:digit:]a-c[.NUL.]]`

エスケープ

POSIX 標準は、POSIX 拡張正規表現についてエスケープシーケンスを定義していない。ただし、以下の例外がある。

- 特殊文字の前にエスケープが付いている場合は、文字そのものにマッチする。
- 通常の文字の前にエスケープを付けた場合の効果は未定義である。
- 文字クラス宣言内のエスケープは、エスケープ文字自身にマッチする。言い換えると、エスケープ文字は文字クラス宣言内では特殊文字ではない。よって`[\^]`は直値の `"\"` か `"^"` にマッチする。

しかしながら、これではいささか制限が強すぎるため、Boost.Regex は以下の標準互換の拡張をサポートする。

特定の文字にマッチするエスケープ

以下のエスケープシーケンスは、すべて 1 文字の別名である。

エスケープ	文字
`\a`	‘\a’
`\e`	0x1B
`\f`	\f
`\n`	\n
`\r`	\r
`\t`	\t

エスケープ	文字
\v	\v
\b	\b（文字クラス宣言内のみ）
\cX	ASCII エスケープシーケンス。コードポイントが X % 32 の文字
\xdd	16 進エスケープシーケンス。コードポイントが 0xdd の文字にマッチする。
\x{dddd}	16 進エスケープシーケンス。コードポイントが 0xdddd の文字にマッチする。
\0ddd	8 進エスケープシーケンス。コードポイントが 0ddd の文字にマッチする。
\N{name}	シンボル名 *name* の文字にマッチする。例えば\\N{newline}は文字 "\n" にマッチする。

「単一文字」文字クラス

x が文字クラス名である場合、エスケープ文字 *x* はその文字クラスに属するあらゆる文字にマッチし、エスケープ文字 *X* はその文字クラスに属さないあらゆる文字にマッチする。

既定でサポートされているものは以下のとおりである。

エスケープシーケンス	等価な文字クラス
\d	[[:digit:]]
\l	[[:lower:]]
\s	[[:space:]]
\u	[[:upper:]]
\w	[[:word:]]
\D	[^[:digit:]]
\L	[^[:lower]]
\S	[^[:space:]]
\U	[^[:upper:]]
\W	[^[:word:]]

文字プロパティ

次の表の文字プロパティ名はすべて文字クラスで使用する名前と等価である。

形式	説明	等価な文字集合の形式
\pX	プロパティ *X* をもつあらゆる文字にマッチする。	[[:X:]]
\p{Name}	プロパティ *Name* をもつあらゆる文字にマッチする。	[[:Name:]]
\PX	プロパティ *X* をもたないあらゆる文字にマッチする。	[^[:X:]]
\P{Name}	プロパティ *Name* をもたないあらゆる文字にマッチする。	[^[:Name:]]

例えば\pd は\p{digit}と同様、あらゆる「数字」（“digit”）にマッチする。

単語境界

次のエスケープシーケンスは単語の境界にマッチする。

エスケープ	意味
\<	単語の先頭にマッチする。
\>	単語の終端にマッチする。
\b	単語境界（単語の先頭か終端）にマッチする。
\B	単語境界以外にマッチする。

バッファ境界

以下はバッファ境界にのみマッチする。この場合の「バッファ」とは、マッチ対象の入力テキスト全体である（^および$はテキスト中の改行にもマッチすることに注意していただきたい）。

エスケープ	意味
\`	バッファの先頭にのみマッチする。
\'	バッファの終端にのみマッチする。
\A	バッファの先頭にのみマッチする（\`と同じ）。
\z	バッファの終端にのみマッチする（\'と同じ）。
\Z	バッファ終端の長さ 0 以上の改行シーケンスにマッチする。正規表現\n*\zと等価である。

継続エスケープ（Continuation Escape）

シーケンス\Gは最後にマッチが見つかった位置、あるいは前回のマッチが存在しない場合はマッチ対象テキストの先頭にのみマッチする。各マッチが 1 つ前のマッチの終端から始まっているようなマッチをテキスト中から列挙する場合に、このシーケンスは有効である。

クォーティングエスケープ（Quoting Escape）

エスケープシーケンス\Qは「クォートされたシーケンス」の開始を表す。以降、正規表現の終端か\Eまでの文字はすべて直値として扱われる。例えば、正規表現\Q*+\Ea+は以下のいずれかにマッチする。

```
\*+a
\*+aaa
```

Unicode エスケープ

エスケープ	意味
\C	単一のコードポイントにマッチする。Boost.Regex では.演算子とまったく同じ意味である。
\X	結合文字シーケンス（非結合文字に 0 以上の結合文字シーケンスが続く）にマッチする。

その他のエスケープ

その他のエスケープシーケンスは、エスケープ対象の文字そのものにマッチする。例えば`\@`は直値 `"@"` にマッチする。

演算子の優先順位

演算子の優先順位は以下のとおりである。

1. 照合関係の括弧記号 `[==]` `[::]` `[..]`
2. エスケープ `\`
3. 文字集合（括弧式） `[]`
4. グループ `()`
5. 単一文字の繰り返し `*` `+` `?` `{m,n}`
6. 結合
7. アンカー `^$`
8. 選択 `|`

マッチするもの

正規表現のマッチに複数の、可能な「最良」マッチは最左最長の規則で得られるものである。

バリエーション

egrep

`egrep` フラグを設定して正規表現をコンパイルすると、改行区切りの POSIX 拡張正規表現のリストとして扱われ、リスト内にマッチする正規表現があればマッチとなる。例えば次のコードは、

```
boost::regex e("abc\ndef", boost::regex::egrep);
```

POSIX 基本正規表現の `abc` か `def` のいずれかにマッチする。

名前が示すように、この動作は Unix ユーティリティの egrep および grep に-E オプションを付けて使用したものに合致する。

awk

POSIX 拡張機能に加えて、エスケープ文字が文字クラス宣言内で特殊となる。

さらに、POSIX 拡張仕様が定義しないいくつかのエスケープシーケンスをサポートすることが要求される。Boost.Regex はこれらのエスケープシーケンスを既定でサポートする。

オプション

正規表現構築時に `extended` および `egrep` オプションとともに指定可能なフラグが多数ある。特に `collate`、`no_subs`、`icase` オプションが大文字小文字の区別やロカール依存の動作を変更するのに対し、`newline_alt` オプションは構文を変更するという

点に注意していただきたい。

参考

IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Base Definitions and Headers, Section 9, Regular Expressions。

IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Shells and Utilities, Section 4, Utilities, egrep。

IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Shells and Utilities, Section 4, Utilities, awk。

POSIX 基本正規表現構文

概要

POSIX 基本正規表現構文は Unix ユーティリティ `sed` が使用しており、`grep` および `emacs` がその変種を使用している。Boost.Regex で POSIX 基本正規表現を使用するには、コンストラクタにフラグ `basic` を渡す（`syntax_option_type` を見よ）。例えば、

```
// e1 は大文字小文字を区別する POSIX 基本正規表現：
boost::regex e1(my_expression, boost::regex::basic);
// e2 は大文字小文字を区別しない POSIX 基本正規表現：
boost::regex e2(my_expression, boost::regex::basic|boost::regex::icase);
```

POSIX 基本構文

POSIX 基本正規表現では、以下の特別なものを除くあらゆる文字が文字そのものにマッチする。

```
.[\*^$
```

ワイルドカード

文字集合外部の`.`1 文字は、以下以外のあらゆる文字 1 文字にマッチする。

- NULL 文字（マッチアルゴリズムにフラグ `match_not_dot_null` を渡した場合）。
- 改行文字（マッチアルゴリズムにフラグ `match_not_dot_newline` を渡した場合）。

アンカー

`^`は、正規表現の先頭、あるいは部分式の先頭で使用した場合に行頭にマッチする。

`$`は、正規表現の終端、あるいは部分式の終端で使用した場合に行末にマッチする。

マーク済み部分式

開始が\(で終了が\)の節は部分式として機能する。マッチした部分式はすべてマッチアルゴリズムにより個別のフィールドに分けられる。マーク済み部分式は繰り返しと後方参照により参照が可能である。

繰り返し

あらゆるアトム（文字、部分式、文字クラス）は*演算子による繰り返しが可能である。

例えばa*は文字aの0回以上の繰り返しにマッチする（アトムの0回の繰り返しは空文字列にマッチする）ため、正規表現a*bは以下のいずれにもマッチする。

```
b
ab
aaaaaaaab
```

アトムの繰り返しは回数境界指定の繰り返しによっても可能である。

a\{n\}は"a"のちょうどn回の繰り返しにマッチする。

a\{n,\}は"a"のn回以上の繰り返しにマッチする。

a\{n,m\}は"a"のn回以上m回以下の繰り返しにマッチする。

例えば

```
^a\{2,3\}$
```

は、次のいずれにもマッチするが、

```
aa
aaa
```

次のいずれにもマッチしない。

```
a
aaaa
```

直前の構造が繰り返し不能な場合に繰り返し演算子を使うとエラーになる。例えば次は

```
a\(*\)
```

*演算子を適用可能なものがないためエラーとなる。

後方参照

エスケープ文字の直後に数字 *n* があると、部分式 *n* にマッチしたものと同じ文字列にマッチする。*n* は0から9の範囲である。例えば次の正規表現は、

```
^\(a*\).*\1$
```

次の文字列にマッチする。

```
aaabbaaa
```

しかし、次の文字列にはマッチしない。

```
aaabba
```

文字集合

文字集合は`[`で始まり`]`で終わる括弧式であり、文字の集合を定義する。集合に含まれるいずれかの 1 文字にマッチする。

文字集合に含められる要素は以下の組み合わせである。

単一の文字

例えば`[abc]`は `"a"`、`"b"`、`"c"` のいずれか 1 文字にマッチする。

文字範囲

例えば`[a-c]`は ‘a’ から ‘c’ までの範囲の 1 文字にマッチする。POSIX 拡張正規表現の既定では、文字 *x* が *y* から *z* の範囲であるとは、文字の照合順がその範囲内にある場合をいう。結果はロカールの影響を受ける。この動作は正規表現構築時に `collate` オプションフラグを設定しないことで抑止でき、文字が特定の範囲内にあるかどうかは文字のコードポイントのみで決定する。

否定

括弧式が文字`^`で始まっている場合は、正規表現に含まれる文字の補集合となる。例えば`[^a-c]`は範囲 `a-c` を除くあらゆる文字にマッチする。

文字クラス

`[[:name:]]`のような形式の正規表現は名前付き文字クラス「name」にマッチする。例えば`[[:lower:]]`はあらゆる小文字にマッチする。文字クラス名を見よ。

照合要素

`[[.col.]]`のような形式の式は照合要素 *col* にマッチする。照合要素とは、単一の照合単位として扱われる文字か文字シーケンスである。照合要素は範囲の端点としても使用できる。例えば`[[.ae.]-c]`は文字シーケンス “ae” に加えて、現在の範囲 “ae”-c の文字のいずれかにマッチする。 後者において “ae” は現在のロカールにおける単一の照合要素として扱われる。

照合要素は（通常、文字集合内で使用できない）エスケープの代わりとして使用できる。例えば`[[.^.]abc]`は ‘abc^’ のいずれかの 1 文字にマッチする。

この拡張として、照合要素をシンボル名で指定する方法もある。例えば、

```
[[.NUL.]]
```

は NUL 文字にマッチする。照合要素名を見よ。

等価クラス

`[[=col=]]`のような形式の正規表現は、第 1 位のソートキーが照合要素 *col* と同じ文字および照合要素にマッチする。照合要素名 *col* は照合シンボル名でもよい。第 1 位のソートキーでは大文字小文字の違い、アクセント記号の有無、ロカール固有のテーラリング（tailoring）は無視される。よって`[[=a=]]`は a、À、Á、Â、Ã、Ä、Å、A、à、á、â、ã、ä および å のいずれにもマッチする。残念ながらこの機能の実装はプラットフォームの照合と地域化のサポートに依存し、すべてのプラットフォームで移植性の高い動作は期待できず、単一のプラットフォームにおいてもすべてのロカールで動作するとは限らない。

結合

以上の要素はすべて 1 つの文字集合宣言内で結合可能である。例: `[[:digit:]a-c[.NUL.]]`

エスケープ

上で述べた`\{`、`\}`、`\(`および`\)`を例外として、エスケープの直後に文字が現れる場合はその文字にマッチする。これにより特殊文字

```
.[\*^$
```

を「通常の」文字にすることができる。エスケープ文字は文字集合内ではその特殊な意味を失うことに注意していただきたい。したがって`[\^]`は直値の `"\"` か `"^"` にマッチする。

マッチするもの

正規表現のマッチに複数の、可能な「最良」マッチは最左最長の規則で得られるものである。

バリエーション

grep

`grep` フラグを設定して正規表現をコンパイルすると、改行区切りの POSIX 基本正規表現のリストとして扱われ、リスト内にマッチする正規表現があればマッチとなる。例えば次のコードは、

```
boost::regex e("abc\ndef", boost::regex::grep);
```

POSIX 基本正規表現の `abc` か `def` のいずれかにマッチする。

名前が示すように、この動作は Unix ユーティリティの grep に合致する。

emacs

POSIX 基本機能に加えて以下の文字が特殊である。

文字	説明
`+`	直前のアトムの 1 回以上の繰り返し。
`?`	直前のアトムの 0 回か 1 回の繰り返し。
`*?`	`*`の貪欲でない版。
`+?`	`+`の貪欲でない版。
`??`	`?`の貪欲でない版。

また、以下のエスケープシーケンスが考慮される。

エスケープ	説明
`|`	選択を表す。
`\(?: ... )`	マーク付けを行わないグループ構造。余計な部分式を生成することなく、字句的なグループ化が可能である。
`\w`	単語構成文字にマッチする。
`\W`	非単語構成文字にマッチする。
`\sx`	構文グループ *x* に属する文字にマッチする。 次の emacs グルーピングをサポートする：‘s’、‘ ’、‘_’、‘w’、‘.’、‘)’、‘(’、‘"’、‘\’、‘>’、‘<’。 詳細は emacs のドキュメントを見よ。
`\Sx`	構文グループ *x* に属さない文字にマッチする。
`\c`および`\C`	これらはサポートしない。
`\``	バッファ（あるいはマッチ対象テキスト）の先頭 0 文字にのみマッチする。
`\'`	バッファ（あるいはマッチ対象テキスト）の終端 0 文字にのみマッチする。
`\b`	単語境界の先頭 0 文字にのみマッチする。
`\B`	非単語境界の先頭 0 文字にのみマッチする。
`\<`	単語の先頭 0 文字にのみマッチする。
`\>`	単語の終端 0 文字にのみマッチする。

最後に emacs スタイルの正規表現マッチは、Perl の「深さ優先探索」規則にしたがうことに注意していただきたい。emacs の正規表現は、POSIX スタイルの最左最長規則と調和しない Perl ライクの拡張を含むためこのような動作をする。

オプション

正規表現構築時に `basic` および `grep` オプションとともに指定可能なフラグが多数ある。特に `collate`、`icase` オプションが大文字小文字の区別やロカール依存の動作を変更するのに対し、`newline_alt`、`no_char_classes`、`no_intervals`、`bk_plus_qm`、`bk_plus_vbar` オプションは構文を変更するという点に注意していただきたい。

参考

IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Base Definitions and Headers, Section 9, Regular Expressions (FWD.1)。

IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Shells and Utilities, Section 4, Utilities, grep (FWD.1)。

Emacs 21.3。

文字クラス名

常にサポートされている文字クラス

以下の文字クラスが Boost.Regex において常にサポートされている。

名前	POSIX 標準名か	説明
alnum	○	アルファベットか数字。
alpha	○	アルファベット。
blank	○	行区切り以外の空白類文字。
cntrl	○	制御文字。
d	×	10 進数字。
digit	○	10 進数字。
graph	○	グラフィカルな文字。
l	×	小文字。
lower	○	小文字。
print	○	印字可能な文字。
punct	○	区切り文字。
s	×	空白類文字。
space	○	空白類文字。
unicode	○	コードポイントが 256 以上の文字。
u	×	大文字。
upper	○	大文字。
w	×	単語構成文字（アルファベット、数字、アンダースコア）。
word	×	単語構成文字（アルファベット、数字、アンダースコア）。
xdigit	○	16 進数字。

Unicode 正規表現によりサポートされる文字クラス

以下の文字クラスは Unicode 正規表現（`u32regex` 型）でのみサポートされている。使用する名前は Unicode 標準 4 章と同じである。

短い名前	長い名前

(なし)	ASCII
(なし)	Any
(なし)	Assigned
C*	Other
Cc	Control
Cf	Format
Cn	Not Assigned
Co	Private Use
Cs	Surrogate
L*	Letter
Ll	Lowercase Letter
Lm	Modifier Letter
Lo	Other Letter
Lt	Titlecase
Lu	Uppercase Letter
M*	Mark
Mc	Spacing Combining Mark
Me	Enclosing Mark
Mn	Non-Spacing Mark
N*	Number
Nd	Decimal Digit Number
Nl	Letter Number
No	Other Number
P*	Punctuation
Pc	Connector Punctuation
Pd	Dash Punctuation
Pe	Close Punctuation
Pf	Final Punctuation
Pi	Initial Punctuation
Po	Other Punctuation
Ps	Open Punctuation
S*	Symbol
Sc	Currency Symbol
Sk	Modifier Symbol
Sm	Math Symbol
So	Other Symbol
Z*	Separator
Zl	Line Separator
Zp	Paragraph Separator

Zs	Space Paragraph

照合名

二重字

照合名として使用可能な二重字は以下のとおりである。

“ae”、“Ae”、“AE”、“ch”、“Ch”、“CH”、“ll”、“Ll”、“LL”、“ss”、“Ss”、“SS”、“nj”、“Nj”、“NJ”、“dz”、“Dz”、“DZ”、“lj”、“Lj” および “LJ”。

例えば次の正規表現は、

```
[[.ae.]-c]
```

照合順が “ae” と “c” の間となるあらゆる文字にマッチする。

POSIX シンボル名

単一文字に加えて以下の表のシンボル名は照合要素名として利用可能である。これにより `"["` か `"]"` にマッチさせたい場合に、例えば次のように書くことができる。

```
[[.left-square-bracket.][.right-square-bracket.]]
```

名前	文字
NUL	\x00
SOH	\x01
STX	\x02
ETX	\x03
EOT	\x04
ENQ	\x05
ACK	\x06
alert	\x07
backspace	\x08
tab	\t
newline	\n
vertical-tab	\v
form-feed	\f
carriage-return	\r
SO	\xE
SI	\xF
DLE	\x10

名前	文字
DC1	\x11
DC2	\x12
DC3	\x13
DC4	\x14
NAK	\x15
SYN	\x16
ETB	\x17
CAN	\x18
EM	\x19
SUB	\x1A
ESC	\x1B
IS4	\x1C
IS3	\x1D
IS2	\x1E
IS1	\x1F
space	\x20
exclamation-mark	!
quotation-mark	"
number-sign	#
dollar-sign	$
percent-sign	%
ampersand	&
apostrophe	'
left-parenthesis	(
right-parenthesis	)
asterisk	*
plus-sign	+
comma	,
hyphen	-
period	.
slash	/
zero	0
one	1
two	2
three	3
four	4
five	5
six	6

名前	文字
seven	7
eight	8
nine	9
colon	:
semicolon	;
less-than-sign	<
equals-sign	=
greater-than-sign	>
question-mark	?
commercial-at	@
left-square-bracket	[
backslash	\
right-square-bracket	]
circumflex	^
underscore	_
grave-sign	`
left-curly-bracket	{
vertical-line	|
right-curly-bracket	}
tilde	~
DEL	\x7F

名前付き Unicode 文字

（`u32regex` 型を用いて）Unicode 正規表現を使用すると、Unicode 文字の通常のシンボル名（`Unidata.txt` にデータがある）が考慮される。よって、例えば

```
[[.CYRILLIC CAPITAL LETTER I.]]
```

は Unicode 文字 0x0418 にマッチする。

最左最長マッチの規則

POSIX 基本および拡張正規表現では、特定の位置で正規表現のマッチを行う方法が 2 つ以上存在することがよくあり、以下のようにして「最良の」マッチが決定する。

1. 最も左のマッチを検索する。ここでマッチ候補が 1 つだけであれば、それを返す。
2. 最左マッチ候補の中で最長のマッチを検索する。候補が 1 つに絞られれば、それを返す。
3. マーク済み部分式がなければ残りの候補に優劣をつけることはできないので、最初の候補を返す。
4. この時点の候補から、最左位置で 1 番目の部分式にマッチしたマッチを検索する。そのようなマッチが 1 つだけであれば、

それを返す。

5. この時点の候補から、1 番目の部分式に対する最長のマッチを検索する。そのようなマッチが 1 つだけであれば、それを返す。
6. 2 番目以降の部分式について 4 から 5 を繰り返す。
7. マッチの候補が 2 つ以上残っていればそれらに優劣をつけることはできないので、最初の候補を返す。

検索・置換書式化文字列の構文

書式化文字列は、アルゴリズム regex_replace および match_results<>::format で文字列を変換するのに使用する。

書式化文字列には sed、Perl および Boost 拡張の 3 種類がある。

これとは別に、上に挙げた関数にフラグ format_literal を渡すと書式化文字列は直値文字列として扱われ、出力にそのままコピーされる。

sed 書式化文字列の構文

sed スタイルの書式化文字列は、以下以外のすべての文字を直値として扱う。

文字	説明
&	アンパーサンド文字は出力ストリーム中でマッチした正規表現全体に置換される。直値の '&' を出力するには\&を使用する。
\	エスケープシーケンスを指定する。

エスケープ文字の直後に文字 *x* が続いている場合、*x* が以下のエスケープシーケンス以外であればその文字を出力する。

エスケープ	意味
\a	ベル文字 '\a' を出力する。
\e	ANSI エスケープ文字（コードポイント 27）を出力する。
\f	フォームフィード文字 '\f' を出力する。
\n	改行文字 '\n' を出力する。
\r	復改文字 '\r' を出力する。
\t	タブ文字 '\t' を出力する。
\v	垂直タブ文字 '\v' を出力する。
\xDD	16 進数コードポイントが 0xDD である文字を出力する。
\x{DDDD}	16 進数コードポイントが 0xDDDD である文字を出力する。
\cX	ANSI エスケープシーケンス "escape-X" を出力する。
\D	*D* が範囲 1-9 の 10 進数字であれば、部分式 *D* にマッチしたテキストを出力する。

Perl 書式化文字列の構文

Perl スタイルの書式化文字列は、プレースホルダーおよびエスケープシーケンスを開始する '$' および '\' 以外のすべての文字を直値として扱う。

プレースホルダーシーケンスは、正規表現に対するマッチのどの部分を出力に送るかを指定する。

プレースホルダー	意味
`$&`	正規表現全体にマッチした部分を出力する。
`$MATCH`	`$&`と同じ。
`${^MATCH}`	`$&`と同じ。
`$``	最後に見つかったマッチの終端（前回のマッチが存在しない場合はテキストの先頭）から現在のマッチの先頭までのテキストを出力する。
`$PREMATCH`	`$``と同じ。
`${^PREMATCH}`	`$``と同じ。
`$'`	現在のマッチの終端より後方のすべてのテキストを出力する。
`$POSTMATCH`	`$'`と同じ。
`${^POSTMATCH}`	`$'`と同じ。
`$+`	正規表現中の最後のマーク済み部分式にマッチした部分を出力する。
`$LAST_PAREN_MATCH`	`$+`と同じ。
`$LAST_SUBMATCH_RESULT`	最後の部分式に実際にマッチした部分を出力する。
`$^N`	`$LAST_SUBMATCH_RESULT`と同じ。
`$$`	直値の ‘$’ を出力する。
`$n`	*n* 番目の部分式にマッチした部分を出力する。
`${n}`	*n* 番目の部分式にマッチした部分を出力する。
`$+{NAME}`	“NAME” という名前の部分式にマッチした部分を出力する。

上に挙げなかった $ プレースホルダーはすべて直値の ‘$’ として扱われる。

エスケープ文字の直後に文字 *x* が続いている場合、*x* が以下のエスケープシーケンス以外であればその文字を出力する。

エスケープ	意味
`\a`	ベル文字 ‘\a’ を出力する。
`\e`	ANSI エスケープ文字（コードポイント 27）を出力する。
`\f`	フォームフィード文字 ‘\f’ を出力する。
`\n`	改行文字 ‘\n’ を出力する。
`\r`	復改文字 ‘\r’ を出力する。
`\t`	タブ文字 ‘\t’ を出力する。
`\v`	垂直タブ文字 ‘\v’ を出力する。
`\xDD`	16 進数コードポイントが 0xDD である文字を出力する。
`\x{DDDD}`	16 進数コードポイントが 0xDDDD である文字を出力する。
`\cX`	ANSI エスケープシーケンス “escape-X” を出力する。
`\D`	*D* が範囲 1-9 の 10 進数字であれば、部分式 *D* にマッチしたテキストを出力する。
`\l`	次に出力する 1 文字を小文字で出力する。
`\u`	次に出力する 1 文字を大文字で出力する。

エスケープ	意味
\L	以降\Eが現れるまで、出力する文字をすべて小文字にする。
\U	以降\Eが現れるまで、出力する文字をすべて大文字にする。
\E	\Lおよび\Uシーケンスを終了する。

Boost 拡張書式化文字列の構文

Boost 拡張書式化文字列は、'$'、'\'、'('、')'、'?' および '.' 以外のすべての文字を直値として扱う。

グループ化

文字(および)は字句的なグループ化を行う。したがって括弧そのものを出力する場合は\(および\)を使用する。

条件

文字?は条件式を開始する。一般形は、以下である。

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

ただし、*N* は 10 進数字である。

部分式 *N* がマッチした場合、*true-expression* が評価され出力に送られる。それ以外の場合は *false-expression* が評価され出力に送られる。

通常、あいまいさを回避するために条件式を括弧で囲む必要がある。

例えば書式化文字列(?1foo:bar)は、部分式 $1 がマッチした場合は foo で、$2 がマッチした場合は bar で見つかった各マッチを置換する。

添字が 9 より大きい部分式にアクセスする場合は、以下を使用する。

```
?{INDEX}true-expression:false-expression
```

名前付き部分式にアクセスする場合は、以下を使用する。

```
?{NAME}true-expression:false-expression
```

プレースホルダーシーケンス

プレースホルダーシーケンスは、正規表現に対するマッチのどの部分を出力に送るかを指定する。

プレースホルダー	意味
$&	正規表現全体にマッチした部分を出力する。
$MATCH	$&と同じ。
${^MATCH}	$&と同じ。

プレースホルダー	意味
$`	最後に見つかったマッチの終端（前回のマッチが存在しない場合はテキストの先頭）から現在のマッチの先頭までのテキストを出力する。
$PREMATCH	$`と同じ。
${^PREMATCH}	$`と同じ。
$'	現在のマッチの終端より後方のすべてのテキストを出力する。
$POSTMATCH	$'と同じ。
${^POSTMATCH}	$'と同じ。
$+	正規表現中の最後のマーク済み部分式にマッチした部分を出力する。
$LAST_PAREN_MATCH	$+と同じ。
$LAST_SUBMATCH_RESULT	最後の部分式に実際にマッチした部分を出力する。
$^N	$LAST_SUBMATCH_RESULTと同じ。
$$	直値の ‘$’ を出力する。
$n	*n* 番目の部分式にマッチした部分を出力する。
${n}	*n* 番目の部分式にマッチした部分を出力する。
$+{NAME}	“NAME” という名前の部分式にマッチした部分を出力する。

上に挙げなかった $ プレースホルダーはすべて直値の ‘$’ として扱われる。

エスケープシーケンス

エスケープ文字の直後に文字 *x* が続いている場合、*x* が以下のエスケープシーケンス以外であればその文字を出力する。

エスケープ	意味
\a	ベル文字 ‘\a’ を出力する。
\e	ANSI エスケープ文字（コードポイント 27）を出力する。
\f	フォームフィード文字 ‘\f’ を出力する。
\n	改行文字 ‘\n’ を出力する。
\r	復改文字 ‘\r’ を出力する。
\t	タブ文字 ‘\t’ を出力する。
\v	垂直タブ文字 ‘\v’ を出力する。
\xDD	16 進数コードポイントが 0xDD である文字を出力する。
\x{DDDD}	16 進数コードポイントが 0xDDDD である文字を出力する。
\cX	ANSI エスケープシーケンス “escape-X” を出力する。
\D	*D* が範囲 1-9 の 10 進数字であれば、部分式 *D* にマッチしたテキストを出力する。
\l	次に出力する 1 文字を小文字で出力する。
\u	次に出力する 1 文字を大文字で出力する。
\L	以降\Eが現れるまで、出力する文字をすべて小文字にする。

エスケープ	意味
\U	以降\Eが現れるまで、出力する文字をすべて大文字にする。
\E	\Lおよび\Uシーケンスを終了する。

リファレンス

basic_regex

概要

```
#include <boost/regex.hpp>
```

テンプレートクラス basic_regex は、正規表現の解析とコンパイルをカプセル化する。このクラスは 2 つのテンプレート引数をとる。

- charT は文字型を決定する。すなわち char か wchar_t のいずれかである。charT のコンセプトを見よ。
- traits は、例えばどの文字クラス名を考慮するか、といった文字型の振る舞いを決定する。既定の特性クラスとして regex_traits<charT>が用意されている。traits のコンセプトを見よ。

簡単に使用できるように、標準的な basic_regex インスタンスを定義する typedef が 2 つある。カスタムの特性クラスか非標準の文字型（例えば Unicode サポートを見よ）を使用するつもりがなければ、この 2 つだけを使用すればよい。

```
namespace boost{

template <class charT, class traits = regex_traits<charT> >
class basic_regex;

typedef basic_regex<char>      regex;
typedef basic_regex<wchar_t>   wregex;

}
```

以下が basic_regex の定義である。basic_string クラスに基づいており、charT の定数コンテナの要求事項を満足する。

```
namespace boost{

template <class charT, class traits = regex_traits<charT> >
class basic_regex {
   public:
   // 型：
   typedef          charT                              value_type;
   typedef          implementation-specific            const_iterator;
   typedef          const_iterator                     iterator;
   typedef          charT&                             reference;
   typedef          const charT&                       const_reference;
   typedef          std::ptrdiff_t                     difference_type;
   typedef          std::size_t                        size_type;
   typedef          regex_constants::syntax_option_type flag_type;
   typedef typename traits::locale_type                locale_type;

   // 定数：
```

```
   // メインオプションの選択：
   static const regex_constants:: syntax_option_type normal
                                                = regex_constants::normal;
   static const regex_constants:: syntax_option_type ECMAScript
                                                = normal;
   static const regex_constants:: syntax_option_type JavaScript
                                                = normal;
   static const regex_constants:: syntax_option_type Jscript
                                                = normal;
   static const regex_constants:: syntax_option_type basic
                                                = regex_constants::basic;
   static const regex_constants:: syntax_option_type extended
                                                = regex_constants::extended;
   static const regex_constants:: syntax_option_type awk
                                                = regex_constants::awk;
   static const regex_constants:: syntax_option_type grep
                                                = regex_constants::grep;
   static const regex_constants:: syntax_option_type egrep
                                                = regex_constants::egrep;
   static const regex_constants:: syntax_option_type sed
                                                = basic = regex_constants::sed;
   static const regex_constants:: syntax_option_type perl
                                                = regex_constants::perl;
   static const regex_constants:: syntax_option_type literal
                                                = regex_constants::literal;

   // Perl 正規表現固有の修飾子：
   static const regex_constants:: syntax_option_type no_mod_m
                                                = regex_constants::no_mod_m;
   static const regex_constants:: syntax_option_type no_mod_s
                                                = regex_constants::no_mod_s;
   static const regex_constants:: syntax_option_type mod_s
                                                = regex_constants::mod_s;
   static const regex_constants:: syntax_option_type mod_x
                                                = regex_constants::mod_x;

   // POSIX 基本正規表現固有の修飾子：
   static const regex_constants:: syntax_option_type bk_plus_qm
                                                = regex_constants::bk_plus_qm;
   static const regex_constants:: syntax_option_type bk_vbar
                                                = regex_constants::bk_vbar;
   static const regex_constants:: syntax_option_type no_char_classes
                                                = regex_constants::no_char_classes;
   static const regex_constants:: syntax_option_type no_intervals
                                                = regex_constants::no_intervals;

   // 共通の修飾子：
   static const regex_constants:: syntax_option_type nosubs
                                                = regex_constants::nosubs;
   static const regex_constants:: syntax_option_type optimize
                                                = regex_constants::optimize;
   static const regex_constants:: syntax_option_type collate
                                                = regex_constants::collate;
   static const regex_constants:: syntax_option_type newline_alt
                                                = regex_constants::newline_alt;
   static const regex_constants:: syntax_option_type no_except
                                                = regex_constants::newline_alt;

   // 構築、コピー、解体：
```

```
   explicit basic_regex ();
   explicit basic_regex(const charT* p, flag_type f = regex_constants::normal);
   basic_regex(const charT* p1, const charT* p2,
               flag_type f = regex_constants::normal);
   basic_regex(const charT* p, size_type len, flag_type f);
   basic_regex(const basic_regex&);

   template <class ST, class SA>
   explicit basic_regex(const basic_string<charT, ST, SA>& p,
                        flag_type f = regex_constants::normal);

   template <class InputIterator>
   basic_regex(InputIterator first, InputIterator last,
               flag_type f = regex_constants::normal);

   ~basic_regex();
   basic_regex& operator=(const basic_regex&);
   basic_regex& operator= (const charT* ptr);

   template <class ST, class SA>
   basic_regex& operator= (const basic_string<charT, ST, SA>& p);
   // イテレータ：
   std::pair<const_iterator, const_iterator> subexpression(size_type n) const;
   const_iterator begin() const;
   const_iterator end() const;
   // 容量：
   size_type size() const;
   size_type max_size() const;
   bool empty() const;
   unsigned mark_count()const;
   //
   // 変更：
   basic_regex& assign(const basic_regex& that);
   basic_regex& assign(const charT* ptr,
                       flag_type f = regex_constants::normal);
   basic_regex& assign(const charT* ptr, unsigned int len, flag_type f);

   template <class string_traits, class A>
   basic_regex& assign(const basic_string<charT, string_traits, A>& s,
                       flag_type f = regex_constants::normal);

   template <class InputIterator>
   basic_regex& assign(InputIterator first, InputIterator last,
                       flag_type f = regex_constants::normal);

   // const な操作：
   flag_type flags() const;
   int status()const;
   basic_string<charT> str() const;
   int compare(basic_regex&) const;
   // ロカール：
   locale_type imbue(locale_type loc);
   locale_type getloc() const;
   // 値の交換
   void swap(basic_regex&) throw();
};

template <class charT, class traits>
bool operator == (const basic_regex<charT, traits>& lhs,
```

```
                  const basic_regex<charT, traits>& rhs);

template <class charT, class traits>
bool operator != (const basic_regex<charT, traits>& lhs,
                  const basic_regex<charT, traits>& rhs);

template <class charT, class traits>
bool operator < (const basic_regex<charT, traits>& lhs,
                 const basic_regex<charT, traits>& rhs);

template <class charT, class traits>
bool operator <= (const basic_regex<charT, traits>& lhs,
                  const basic_regex<charT, traits>& rhs);

template <class charT, class traits>
bool operator >= (const basic_regex<charT, traits>& lhs,
                  const basic_regex<charT, traits>& rhs);

template <class charT, class traits>
bool operator > (const basic_regex<charT, traits>& lhs,
                 const basic_regex<charT, traits>& rhs);

template <class charT, class io_traits, class re_traits>
basic_ostream<charT, io_traits>&
operator << (basic_ostream<charT, io_traits>& os,
             const basic_regex<charT, re_traits>& e);

template <class charT, class traits>
void swap(basic_regex<charT, traits>& e1,
          basic_regex<charT, traits>& e2);

typedef basic_regex<char> regex;
typedef basic_regex<wchar_t> wregex;

} // namespace boost
```

説明

basic_regex クラスは以下の公開メンバをもつ。

```
// メインオプションの選択：
static const regex_constants:: syntax_option_type normal
                                          = regex_constants::normal;
static const regex_constants:: syntax_option_type ECMAScript
                                          = normal;
static const regex_constants:: syntax_option_type JavaScript
                                          = normal;
static const regex_constants:: syntax_option_type Jscript
                                          = normal;
static const regex_constants:: syntax_option_type basic
                                          = regex_constants::basic;
static const regex_constants:: syntax_option_type extended
                                          = regex_constants::extended;
static const regex_constants:: syntax_option_type awk
                                          = regex_constants::awk;
static const regex_constants:: syntax_option_type grep
                                          = regex_constants::grep;
static const regex_constants:: syntax_option_type egrep
```

```
                                        = regex_constants::egrep;
static const regex_constants:: syntax_option_type sed
                                        = basic = regex_constants::sed;
static const regex_constants:: syntax_option_type perl
                                        = regex_constants::perl;
static const regex_constants:: syntax_option_type literal
                                        = regex_constants::literal;

// Perl 正規表現固有の修飾子：
static const regex_constants:: syntax_option_type no_mod_m
                                        = regex_constants::no_mod_m;
static const regex_constants:: syntax_option_type no_mod_s
                                        = regex_constants::no_mod_s;
static const regex_constants:: syntax_option_type mod_s
                                        = regex_constants::mod_s;
static const regex_constants:: syntax_option_type mod_x
                                        = regex_constants::mod_x;

// POSIX 基本正規表現固有の修飾子：
static const regex_constants:: syntax_option_type bk_plus_qm
                                        = regex_constants::bk_plus_qm;
static const regex_constants:: syntax_option_type bk_vbar
                                        = regex_constants::bk_vbar;
static const regex_constants:: syntax_option_type no_char_classes
                                        = regex_constants::no_char_classes;
static const regex_constants:: syntax_option_type no_intervals
                                        = regex_constants::no_intervals;

// 共通の修飾子：
static const regex_constants:: syntax_option_type nosubs
                                        = regex_constants::nosubs;
static const regex_constants:: syntax_option_type optimize
                                        = regex_constants::optimize;
static const regex_constants:: syntax_option_type collate
                                        = regex_constants::collate;
static const regex_constants:: syntax_option_type newline_alt
                                        = regex_constants::newline_alt;
static const regex_constants:: syntax_option_type no_except
                                        = regex_constants::newline_alt;
```

これらのオプションの意味は syntax_option_type の節にある。

静的定数メンバは名前空間 boost::regex_constants 内で宣言した定数の別名として提供している。名前空間 boost::regex_constants 内で宣言されている syntax_option_type 型の各定数については、basic_regex のスコープで同じ名前・型・値で宣言している。

```
basic_regex();
```

効果: basic_regex クラスのオブジェクトを構築する。

表 1. basic_regex デフォルトコンストラクタの事後条件

要素	値
empty()	true
size()	0
str()	basic_string<charT>()

```
basic_regex(const chartT* p, flag_type f = regex_constants::normal);
```

要件: *p* は null ポインタ以外。

例外: *p* が正しい正規表現でない場合 bad_expression (*f* にフラグ no_except が設定されていない場合)。

効果: basic_regex クラスのオブジェクトを構築する。*f* で指定したオプションフラグにしたがって null 終端文字列 *p* の正規表現を解釈し、オブジェクトの内部有限状態マシンを構築する。

表 2. basic_regex コンストラクタの事後条件

要素	値
empty()	false
size()	char_traits<charT>::length(p)
str()	basic_string<charT>(p)
flags()	*f*
mark_count()	正規表現中に含まれるマーク済み部分式の総数

```
basic_regex(const charT* p1, const charT* p2,
            flag_type f = regex_constants::normal);
```

要件: *p1* と *p2* は null ポインタ以外、かつ p1 < p2。

例外: [*p1*,*p2*) が正しい正規表現でない場合 bad_expression (*f* に no_except が設定されていない場合)。

効果: クラス basic_regex のオブジェクトを構築する。*f* で指定したオプションフラグにしたがって文字シーケンス [p1,p2) の正規表現を解釈し、オブジェクトの内部有限状態マシンを構築する。

表 3. basic_regex コンストラクタの事後条件

要素	値
empty()	false
size()	std::distance(p1,p2)
str()	basic_string<charT>(p1,p2)
flags()	*f*
mark_count()	正規表現中に含まれるマーク済み部分式の総数

```
basic_string(const charT* p, size_type len, flag_type f);
```

要件: *p* は null ポインタ以外、かつ `len < max_size()`。

例外: *p* が正しい正規表現でない場合 `bad_expression`（*f* に `no_except` が設定されていない場合）。

効果: クラス `basic_regex` のオブジェクトを構築する。*f* で指定したオプションフラグにしたがって文字シーケンス [p,p+len) の正規表現を解釈し、オブジェクトの内部有限状態マシンを構築する。

表 4. `basic_regex` コンストラクタの事後条件

要素	値
`empty()`	`false`
`size()`	*len*
`str()`	`basic_string<charT>(p, len)`
`flags()`	*f*
`mark_count()`	正規表現中に含まれるマーク済み部分式の総数

```
basic_regex(const basic_regex& e);
```

効果: オブジェクト *e* をコピーしてクラス `basic_regex` オブジェクトを構築する。

```
template <class ST, class SA>
basic_regex(const basic_string<charT, ST, SA>& s,
            type_flag f = regex_constants::normal);
```

例外: *s* が正しい正規表現でない場合 `bad_expression`（*f* に `no_except` が設定されていない場合）。

効果: `basic_regex` クラスのオブジェクトを構築する。*f* で指定したオプションフラグにしたがって文字列 *s* の正規表現を解釈し、オブジェクトの内部有限状態マシンを構築する。

表 5. `basic_regex` コンストラクタの事後条件

要素	値
`empty()`	`false`
`size()`	`s.size()`
`str()`	*s*
`flags()`	*f*
`mark_count()`	正規表現中に含まれるマーク済み部分式の総数

```
template <class ForwardIterator>
basic_regex(ForwardIterator first, ForwardIterator last,
            flag_type f = regex_constants::normal);
```

例外: [first,last) が正しい正規表現でない場合 `bad_expression`（*f* に `no_except` が設定されていない場合）。

効果: `basic_regex` クラスのオブジェクトを構築する。*f* で指定したオプションフラグにしたがって文字シーケンス [first,last) の正規表現を解釈し、オブジェクトの内部有限状態マシンを構築する。

表 6. basic_regex コンストラクタの事後条件

要素	値
empty()	false
size()	distance(first,last)
str()	basic_string<charT>(first,last)
flags()	*f*
mark_count()	正規表現中に含まれるマーク済み部分式の総数

```
basic_regex& operator=(const basic_regex& e);
```

効果: assign(e.str(), e.flags())の結果を返す。

```
basic_regex& operator=(const charT* ptr);
```

要件: *ptr* は null ポインタ以外。

効果: assign(ptr)の結果を返す。

```
template <class ST, class SA>
basic_regex& operator=(const basic_string<charT, ST, SA>& p);
```

効果: assign(p)の結果を返す。

```
std::pair<const_iterator, const_iterator> subexpression(size_type n) const;
```

効果: 元の正規表現文字列内のマーク済み部分式 *n* の位置を表すイテレータのペアを返す。戻り値のイテレータは begin()および end()からの相対位置である。

要件: 正規表現は syntax_option_type save_subexpression_location を設定してコンパイルしていなければならない。引数 *n* は 0 <= n < mark_count()の範囲になければならない。

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

効果: 正規表現を表す文字シーケンスの開始イテレータを返す。

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

効果: 正規表現を表す文字シーケンスの終了イテレータを返す。

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

効果: 正規表現を表す文字シーケンスの長さを返す。

```
size_type max_size() const;
```

効果: 正規表現を表す文字シーケンスの最大長さを返す。

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

効果: オブジェクトが正しい正規表現を保持していない場合に真を返す。それ以外の場合は偽を返す。

```
unsigned mark_count() const;
```

効果: 正規表現中のマーク済み部分式の数を返す。

```
basic_regex& assign(const basic_regex& that);
```

効果: `assign(that.str(), that.flags())`を返す。

```
basic_regex& assign(const charT* ptr, flag_type f = regex_constants::normal);
```

効果: `assign(string_type(ptr), f)`を返す。

```
basic_regex& assign(const charT* ptr, unsigned int len, flag_type f);
```

効果: `assign(string_type(ptr, len), f)`を返す。

```
template <class string_traits, class A>
basic_regex& assign(const basic_string<charT, string_traits, A>& s,
                    flag_type f = regex_constants::normal);
```

例外: *s* が正しい正規表現でない場合 bad_expression（*f* に `no_except` が設定されていない場合）。

戻り値: `*this`。

効果: *f* で指定したオプションフラグにしたがって文字列 *s* の正規表現を解釈し代入する。

表 7. `basic_regex::assign` の事後条件

要素	値
`empty()`	`false`
`size()`	`s.size()`
`str()`	*s*
`flags()`	*f*
`mark_count()`	正規表現中に含まれるマーク済み部分式の総数

```
template <class InputIterator>
basic_regex& assign(InputIterator first, InputIterator last,
```

```
                    flag_type f = regex_constants::normal);
```

要件: InputIterator 型は入力イテレータの要件(24.1.1)を満たす。

効果: assign(string_type(first, last), f)を返す。

```
flag_type flags() const;
```

効果: オブジェクトのコンストラクタ、あるいは最後の assign の呼び出しで渡した正規表現構文のフラグのコピーを返す。

```
int status() const;
```

効果: 正規表現が正しい正規表現であれば 0、それ以外の場合はエラーコードを返す。このメンバ関数は例外処理を使用できない環境のために用意されている。

```
basic_string<charT> str() const;
```

効果: オブジェクトのコンストラクタ、あるいは最後の assign の呼び出しで渡した文字シーケンスのコピーを返す。

```
int compare(const basic_regex& e) const;
```

効果: flags() == e.flags() であれば str().compare(e.str()) を、それ以外の場合は flags() - e.flags() を返す。

```
locale_type imbue(locale_type l);
```

効果: traits_inst.imbue(l) の結果を返す。traits_inst はオブジェクト内の、テンプレート引数 traits のインスタンス(をデフォルトコンストラクタで初期化したもの)である。

事後条件: empty() == true。

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

効果: traits_inst.getloc() の結果を返す。traits_inst はオブジェクト内の、テンプレート引数 traits のインスタンス(をデフォルトコンストラクタで初期化したもの)である。

```
void swap(basic_regex& e) throw();
```

効果: 2 つの正規表現の内容を交換する。

事後条件: *this は *e* にあった正規表現を保持し、*e* は *this にあった正規表現を保持する。

計算量: 一定。

注意

basic_regex オブジェクト間の比較は実験的なものである。Technical Report on C++ Libraries には記述がなく、basic_regex の他の実装に移植する必要がある場合は注意していただきたい。

```
template <class charT, class traits>
bool operator == (const basic_regex<charT, traits>& lhs,
                  const basic_regex<charT, traits>& rhs);
```

効果: lhs.compare(rhs) == 0 を返す。

```
template <class charT, class traits>
bool operator != (const basic_regex<charT, traits>& lhs,
                  const basic_regex<charT, traits>& rhs);
```

効果: lhs.compare(rhs) != 0 を返す。

```
template <class charT, class traits>
bool operator < (const basic_regex<charT, traits>& lhs,
                 const basic_regex<charT, traits>& rhs);
```

効果: lhs.compare(rhs) < 0 を返す。

```
template <class charT, class traits>
bool operator <= (const basic_regex<charT, traits>& lhs,
                  const basic_regex<charT, traits>& rhs);
```

効果: lhs.compare(rhs) <= 0 を返す。

```
template <class charT, class traits>
bool operator >= (const basic_regex<charT, traits>& lhs,
                  const basic_regex<charT, traits>& rhs);
```

効果: lhs.compare(rhs) >= 0 を返す。

```
template <class charT, class traits>
bool operator > (const basic_regex<charT, traits>& lhs,
                 const basic_regex<charT, traits>& rhs);
```

効果: lhs.compare(rhs) > 0 を返す。

注意

basic_regex のストリーム挿入子は実験的なものであり、正規表現のテキスト表現をストリームに出力する。

```
template <class charT, class io_traits, class re_traits>
basic_ostream<charT, io_traits>&
operator << (basic_ostream<charT, io_traits>& os
             const basic_regex<charT, re_traits>& e);
```

効果: `(os << e.str())`を返す。

```
template <class charT, class traits>
void swap(basic_regex<charT, traits>& lhs,
          basic_regex<charT, traits>& rhs);
```

効果: `lhs.swap(rhs)`を呼び出す。

match_results

概要

```
#include <boost/regex.hpp>
```

正規表現が他の多くの単純なパターンマッチアルゴリズムと異なるのは、マッチを発見するだけでなく、部分式のマッチを生成する点である。各部分式はパターン中の括弧の組`(...)`により、その範囲が与えられる。部分式のマッチをユーザに知らせるために何らかの方法が必要である。部分式マッチの添字付きコレクションとして振舞う`match_results`クラスの定義がそれであり、各部分式マッチは`sub_match`型オブジェクトに含まれる。

テンプレートクラス`match_results`は、正規表現マッチの結果を表す文字シーケンスのコレクションを表現する。`match_results`型のオブジェクトは`regex_match`および`regex_search`アルゴリズムに渡して使用する。またイテレータ`regex_iterator`がこのオブジェクトを返す。このコレクションが使用するストレージは、`match_results`のメンバ関数が必要に応じて割り当て、解放する。

テンプレートクラス`match_results`は（lib.sequence.reqmts）が規定するSequenceの要件を満たす。ただしconst限定の操作に限られる。

大抵の場合、クラステンプレート`match_results`を使用するときは、そのtypedefである`cmatch`、`wcmatch`、`smatch`および`wsmatch`のいずれかを用いる。

```
template <class BidirectionalIterator,
          class Allocator = std::allocator<sub_match<BidirectionalIterator> >
class match_results;

typedef match_results<const char*>              cmatch;
typedef match_results<const wchar_t*>           wcmatch;
typedef match_results<string::const_iterator>   smatch;
typedef match_results<wstring::const_iterator>  wsmatch;

template <class BidirectionalIterator,
          class Allocator = std::allocator<sub_match<BidirectionalIterator> >
class match_results
```

```
{
public:
   typedef          sub_match<BidirectionalIterator>                        value_type;
   typedef          const value_type&                                       const_reference;
   typedef          const_reference                                         reference;
   typedef          implementation defined                                  const_iterator;
   typedef          const_iterator                                          iterator;
   typedef typename iterator_traits<BidirectionalIterator>::difference_type difference_type;
   typedef typename Allocator::size_type                                    size_type;
   typedef          Allocator                                               allocator_type;
   typedef typename iterator_traits<BidirectionalIterator>::value_type      char_type;
   typedef          basic_string<char_type>                                 string_type;

   // 構築、コピー、解体：
   explicit match_results(const Allocator& a = Allocator());
   match_results(const match_results& m);
   match_results& operator=(const match_results& m);
   ~match_results();

   // サイズ：
   size_type size() const;
   size_type max_size() const;
   bool empty() const;
   // 要素アクセス：
   difference_type length(int sub = 0) const;
   template <class charT>
   difference_type length(const charT* sub) const;
   template <class charT, class Traits, class A>
   difference_type length(const std::basic_string<charT, Traits, A>& sub) const;
   difference_type position(unsigned int sub = 0) const;
   difference_type position(const char_type* sub) const;
   template <class charT>
   difference_type position(const charT* sub) const;
   template <class charT, class Traits, class A>
   difference_type position(const std::basic_string<charT, Traits, A>& sub) const;
   string_type str(int sub = 0) const;
   string_type str(const char_type* sub)const;
   template <class Traits, class A>
   string_type str(const std::basic_string<char_type, Traits, A>& sub)const;
   template <class charT>
   string_type str(const charT* sub)const;
   template <class charT, class Traits, class A>
   string_type str(const std::basic_string<charT, Traits, A>& sub)const;
   const_reference operator[](int n) const;
   const_reference operator[](const char_type* n) const;
   template <class Traits, class A>
   const_reference operator[](const std::basic_string<char_type, Traits, A>& n) const;
   template <class charT>
   const_reference operator[](const charT* n) const;
   template <class charT, class Traits, class A>
   const_reference operator[](const std::basic_string<charT, Traits, A>& n) const;

   const_reference prefix() const;
   const_reference suffix() const;
   const_iterator begin() const;
   const_iterator end() const;

   // 書式化：
   template <class OutputIterator, class Formatter>
   OutputIterator format(OutputIterator out,
```

```
                        Formatter& fmt,
                        match_flag_type flags = format_default) const;
   template <class Formatter>
   string_type format(const Formatter fmt,
                      match_flag_type flags = format_default) const;

   allocator_type get_allocator() const;
   void swap(match_results& that);

#ifdef BOOST_REGEX_MATCH_EXTRA
   typedef typename value_type::capture_sequence_type capture_sequence_type;
   const capture_sequence_type& captures(std::size_t i)const;
#endif

};

template <class BidirectionalIterator, class Allocator>
bool operator == (const match_results<BidirectionalIterator, Allocator>& m1,
                  const match_results<BidirectionalIterator, Allocator>& m2);
template <class BidirectionalIterator, class Allocator>
bool operator != (const match_results<BidirectionalIterator, Allocator>& m1,
                  const match_results<BidirectionalIterator, Allocator>& m2);

template <class charT, class traits, class BidirectionalIterator, class Allocator>
basic_ostream<charT, traits>&
   operator << (basic_ostream<charT, traits>& os,
                const match_results<BidirectionalIterator, Allocator>& m);

template <class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
          match_results<BidirectionalIterator, Allocator>& m2);
```

説明

match_results のすべてのコンストラクタにおける Allocator 引数のコピーは、オブジェクトの生涯にわたってコンストラクタとメンバ関数によるメモリ割り当てに使用される。

```
match_results(const Allocator& a = Allocator());
```

効果: match_results クラスのオブジェクトを構築する。この関数の事後条件は次の表のとおりである。

要素	値
empty()	true
size()	0
str()	basic_string<charT>()

```
match_results(const match_results& m);
```

効果: *m* をコピーして match_results クラスのオブジェクトを構築する。

```
match_results& operator=(const match_results& m);
```

効果: *m* を*this に代入する。この関数の事後条件は次の表のとおりである。

要素	値
empty()	m.empty()
size()	m.size()
str(n)	n < m.size() であるすべての整数で m.str(n)
prefix()	m.prefix()
suffix()	m.suffix()
(*this)[n]	n < m.size() であるすべての整数で m[n]
length(n)	n < m.size() であるすべての整数で m.length(n)
position(n)	n < m.size() であるすべての整数で m.position(n)

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

効果: *this 中の sub_match 要素数を返す。これは正規表現中でマッチしたマーク済み部分式の数に 1 を足したものである。

```
size_type max_size()const;
```

効果: *this に格納可能な sub_match 要素の最大数を返す。

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

効果: size() == 0 を返す。

```
difference_type length(int sub = 0)const;
difference_type length(const char_type* sub)const;
template <class charT>
difference_type length(const charT* sub)const;
template <class charT, class Traits, class A>
difference_type length(const std::basic_string<charT, Traits, A>& sub)const;
```

効果: 部分式 *sub* の長さを返す。(*this)[sub].length()と同じである。

文字列を引数に取る多重定義は *n* 番目の名前付き部分式を参照する。指定した名前をもつ部分式がない場合は 0 を返す。

この関数のテンプレート多重定義に渡す文字列・文字の型は、オブジェクトが保持するシーケンスや正規表現の文字型と異なっていてもよい。この場合、文字列は正規表現が保持する文字型に変換される。引数の文字型が正規表現が保持するシーケンスの文字型より幅が大きい場合はコンパイルエラーとなる。これらの多重定義は、マッチを行う正規表現の文字型が Unicode 文字型のような変り種の場合であっても、通常の幅の小さい C 文字列リテラルを引数として渡せるようにしてある。

```
difference_type position(unsigned int sub = 0)const;
difference_type position(const char_type* sub)const;
template <class charT>
difference_type position(const charT* sub)const;
template <class charT, class Traits, class A>
```

```
difference_type position(const std::basic_string<charT, Traits, A>& sub)const;
```

効果: 部分式 *sub* の開始位置を返す。*sub* がマッチしなかった場合は-1 を返す。部分マッチの場合は `(*this)[0].matched` は偽であるが、`position()` は部分マッチの位置を返す。

文字列を引数に取る多重定義は *n* 番目の名前付き部分式を参照する。指定した名前をもつ部分式がない場合は-1 を返す。

この関数のテンプレート多重定義に渡す文字列・文字の型は、オブジェクトが保持するシーケンスや正規表現の文字型と異なっていてもよい。この場合、文字列は正規表現が保持する文字型に変換される。引数の文字型が正規表現が保持するシーケンスの文字型より幅が大きい場合はコンパイルエラーとなる。これらの多重定義は、マッチを行う正規表現の文字型が Unicode 文字型のような変り種の場合であっても、通常の幅の小さい C 文字列リテラルを引数として渡せるようにしてある。

```
string_type str(int sub = 0)const;
string_type str(const char_type* sub)const;
template <class Traits, class A>
string_type str(const std::basic_string<char_type, Traits, A>& sub)const;
template <class charT>
string_type str(const charT* sub)const;
template <class charT, class Traits, class A>
string_type str(const std::basic_string<charT, Traits, A>& sub)const;
```

効果: 部分式 *sub* の文字列を返す。`string_type((*this)[sub])` と同じである。

文字列を引数に取る多重定義は *n* 番目の名前付き部分式を参照する。指定した名前をもつ部分式がない場合は空文字列を返す。

この関数のテンプレート多重定義に渡す文字列・文字の型は、オブジェクトが保持するシーケンスや正規表現の文字型と異なっていてもよい。この場合、文字列は正規表現が保持する文字型に変換される。引数の文字型が正規表現が保持するシーケンスの文字型より幅が大きい場合はコンパイルエラーとなる。これらの多重定義は、マッチを行う正規表現の文字型が Unicode 文字型のような変り種の場合であっても、通常の幅の小さい C 文字列リテラルを引数として渡せるようにしてある。

```
const_reference operator[](int n)const;
const_reference operator[](const char_type* n) const;
template <class Traits, class A>
const_reference operator[](const std::basic_string<char_type, Traits, A>& n) const;
template <class charT>
const_reference operator[](const charT* n) const;
template <class charT, class Traits, class A>
const_reference operator[](const std::basic_string<charT, Traits, A>& n) const;
```

効果: マーク済み部分式 *n* にマッチした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。`n == 0` の場合は、正規表現全体にマッチした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。*n* が範囲外であるかマッチしなかった部分式を指している場合は、`matched` メンバが偽である sub_match オブジェクトを返す。

文字列を引数に取る多重定義は *n* 番目の名前付き部分式にマッチした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。指定した名前をもつ部分式がない場合は `matched` メンバが偽である sub_match オブジェクトを返す。

この関数のテンプレート多重定義に渡す文字列・文字の型は、オブジェクトが保持するシーケンスや正規表現の文字型と異なっていてもよい。この場合、文字列は正規表現が保持する文字型に変換される。引数の文字型が正規表現が保持するシーケンスの

文字型より幅が大きい場合はコンパイルエラーとなる。これらの多重定義は、マッチを行う正規表現の文字型が Unicode 文字型のような変り種の場合であっても、通常の幅の小さい C 文字列リテラルを引数として渡せるようにしてある。

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

効果: マッチ・検索を行う文字列の先頭から見つかったマッチの先頭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。

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

効果: 見つかったマッチの終端からマッチ・検索を行う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。

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

効果: *this に格納されたすべてのマーク済み部分式を列挙する開始イテレータを返す。

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

効果: *this に格納されたすべてのマーク済み部分式を列挙する終了イテレータを返す。

```
template <class OutputIterator, class Formatter>
OutputIterator format(OutputIterator out,
                      Formatter fmt,
                      match_flag_type flags = format_default);
```

要件: 型 OutputIterator が出力イテレータの要件（C++標準 24.1.2）を満たす。

型 Formatter は char_type[] 型の null 終端文字列へのポインタ、char_type 型のコンテナ（例えば std::basic_string<char_type>）、あるいは関数呼び出しにより置換文字列を生成する単項・二項・三項関数子のいずれかでなければならない。関数子の場合、fmt(*this) は置換テキストと使用する char_type のコンテナを返さなければならず、fmt(*this, out) および fmt(*this, out, flags) はいずれも置換テキストを*out に出力し OutputIterator の新しい位置を返さなければならない。

効果: *fmt* が null 終端文字列か char_type のコンテナであれば、文字シーケンス [fmt.begin(), fmt.end()) を OutputIterator *out* にコピーする。*fmt* 中の各書式指定子とエスケープシーケンスは、シーケンスをそれぞれが表す文字（列）か、参照する*this 中の文字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指定子・エスケープシーケンスを使用するか決定し、既定では ECMA-262、ECMAScript 言語仕様、15 章 5.4.11 String.prototype.replace で使用されている書式である。

fmt が関数オブジェクトであれば、関数オブジェクトが受け取った引数の数により以下のようになる。

- fmt(*this) を呼び出し、結果を OutputIterator *out* にコピーする。
- fmt(*this, out) を呼び出す。

- `fmt(*this, out, flags)`を呼び出す。

すべての場合で`OutputIterator`の新しい位置が返される。

詳細は書式化構文ガイドを見よ。

戻り値:*out*。

```
template<class Formatter>
string_type format(Formatter fmt,
                   match_flag_type flags = format_default);
```

要件:型`Formatter`は`char_type[]`型のnull終端文字列へのポインタ、`char_type`型のコンテナ（例えば`std::basic_string<char_type>`）、あるいは関数呼び出しにより置換文字列を生成する単項・二項・三項関数子のいずれかでなければならない。関数子の場合、`fmt(*this)`は置換テキストと使用する`char_type`のコンテナを返さなければならず、`fmt(*this, out)`および`fmt(*this, out, flags)`はいずれも置換テキストを`*out`に出力し`OutputIterator`の新しい位置を返さなければならない。

効果:*fmt*がnull終端文字列か`char_type`のコンテナであれば、文字列*fmt*をコピーする。*fmt*中の各書式指定子とエスケープシーケンスは、シーケンスをそれぞれが表す文字（列）か、参照する`*this`中の文字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指定子・エスケープシーケンスを使用するか決定し、既定ではECMA-262、ECMAScript言語仕様、15章5.4.11 String.prototype.replaceで使用されている書式である。

*fmt*が関数オブジェクトであれば、関数オブジェクトが受け取った引数の数により以下のようになる。

- `fmt(*this)`を呼び出し、結果を返す。
- `fmt(*this, unspecified-output-iterator)`を呼び出す。`unspecified-output-iterator`は出力を結果文字列にコピーする指定なしの`OutputIterator`型である。
- `fmt(*this, unspecified-output-iterator, flags)`を呼び出す。`unspecified-output-iterator`は出力を結果文字列にコピーする指定なしの`OutputIterator`型である。

すべての場合で`OutputIterator`の新しい位置が返される。

詳細は書式化構文ガイドを見よ。

```
allocator_type get_allocator()const;
```

効果:オブジェクトのコンストラクタで渡した`Allocator`のコピーを返す。

```
void swap(match_results& that);
```

効果:2つのシーケンスの内容を交換する。

事後条件:`*this`は、*that*が保持していた、部分式にマッチしたシーケンスを保持する。*that*は、`*this`が保持していた、部分式にマッチしたシーケンスを保持する。

計算量:一定。

```
typedef typename value_type::capture_sequence_type capture_sequence_type;
```

標準ライブラリ Sequence の要件（21.1.1 および表 68 の操作）を満たす実装固有の型を定義する。その `value_type` は `sub_match<BidirectionalIterator>`である。この型が `std::vector<sub_match<BidirectionalIterator> >`となる可能性もあるが、それに依存すべきではない。

```
const capture_sequence_type& captures(std::size_t i)const;
```

効果:部分式 *i* に対するすべての捕捉を格納したシーケンスを返す。

戻り値: `(*this)[i].captures();`

事前条件: `BOOST_REGEX_MATCH_EXTRA` を使ってライブラリをビルドしていなければ、このメンバ関数は定義されない。また正規表現マッチ関数（regex_match、regex_search、regex_iterator、regex_token_iterator）にフラグ `match_extra` を渡していなければ、有用な情報を返さない。

根拠:この機能を有効にするといくつか影響がある。

- `sub_match` がより多くのメモリを占有し、複雑な正規表現をマッチする場合にすぐにメモリやスタック空間の不足に陥る。
- `match_extra` を使用しない場合であっても、処理する機能（例えば独立部分式）によってはマッチアルゴリズムの効率が落ちる。
- `match_extra` を使用するとさらに効率が落ちる（速度が低下する）。ほとんどの場合、さらに必要なメモリ割り当てが起こる。

```
template <class BidirectionalIterator, class Allocator>
bool operator == (const match_results<BidirectionalIterator, Allocator>& m1,
                  const match_results<BidirectionalIterator, Allocator>& m2);
```

効果:2 つのシーケンスの等価性を比較する。

```
template <class BidirectionalIterator, class Allocator>
bool operator != (const match_results<BidirectionalIterator, Allocator>& m1,
                  const match_results<BidirectionalIterator, Allocator>& m2);
```

効果:2 つのシーケンスの非等価性を比較する。

```
template <class charT, class traits, class BidirectionalIterator, class Allocator>
basic_ostream<charT, traits>&
   operator << (basic_ostream<charT, traits>& os,
                const match_results<BidirectionalIterator, Allocator>& m);
```

効果: `os << m.str()` の要領で *m* の内容をストリーム *os* に書き込む。*os* を返す。

```
template <class BidirectionalIterator, class Allocator>
```

```
void swap(match_results<BidirectionalIterator, Allocator>& m1,
          match_results<BidirectionalIterator, Allocator>& m2);
```

効果:2 つのシーケンスの内容を交換する。

sub_match

```
#include <boost/regex.hpp>
```

正規表現が他の多くの単純なパターンマッチアルゴリズムと異なるのは、マッチを発見するだけでなく、部分式のマッチを生成する点である。各部分式はパターン中の括弧の組 `(...)` により、その範囲が与えられる。部分式マッチをユーザに知らせるために何らかの方法が必要である。部分式マッチの添字付きコレクションとして振舞う match_results クラスの定義がそれであり、各部分式マッチは sub_match 型オブジェクトが保持する。

sub_match 型のオブジェクトは match_results 型のオブジェクトの配列要素としてのみ取得可能である。

sub_match 型のオブジェクトは `std::basic_string`、`const charT*`、`const charT` 型のオブジェクトと比較可能である。

sub_match 型のオブジェクトは `std::basic_string`、`const charT*`、`const charT` 型のオブジェクトに追加して新しい `std::basic_string` オブジェクトを生成可能である。

sub_match 型のオブジェクトで示されるマーク済み部分式が正規表現マッチに関与していれば `matched` メンバは**真**と評価され、メンバ `first` と `second` はマッチを形成する文字範囲 [first,second) を示す。それ以外の場合は `matched` は**偽**であり、メンバ `first` と `second` は未定義の値となる。

sub_match 型のオブジェクトで示されるマーク済み部分式が繰り返しになっている場合、その sub_match オブジェクトが表現するのは**最後の**繰り返しに対応するマッチである。すべての繰り返しに対応するすべての捕捉の完全なセットは `captures()` メンバ関数でアクセス可能である（効率に関して深刻な問題があり、この機能は明示的に有効にしなければならない）。

sub_match 型のオブジェクトが部分式 0（マッチ全体）を表現する場合、メンバ `matched` は常に**真**である。ただし正規表現アルゴリズムにフラグ `match_partial` を渡して結果が部分マッチとなる場合はこの限りではなく、メンバ `matched` は**偽**、メンバ `first` と `second` は部分マッチを形成する文字範囲を表現する。

```
namespace boost{

template <class BidirectionalIterator>
class sub_match;

typedef sub_match<const char*>                  csub_match;
typedef sub_match<const wchar_t*>               wcsub_match;
typedef sub_match<std::string::const_iterator>  ssub_match;
typedef sub_match<std::wstring::const_iterator> wssub_match;

template <class BidirectionalIterator>
class sub_match : public std::pair<BidirectionalIterator, BidirectionalIterator>
{
public:
   typedef typename iterator_traits<BidirectionalIterator>::value_type       value_type;
```

```
   typedef typename iterator_traits<BidirectionalIterator>::difference_type  difference_type;
   typedef          BidirectionalIterator                                    iterator;

   bool matched;

   difference_type length()const;
   operator basic_string<value_type>()const;
   basic_string<value_type> str()const;

   int compare(const sub_match& s)const;
   int compare(const basic_string<value_type>& s)const;
   int compare(const value_type* s)const;
#ifdef BOOST_REGEX_MATCH_EXTRA
   typedef implementation-private capture_sequence_type;
   const capture_sequence_type& captures()const;
#endif
};
//
// sub_match 同士の比較：
//
template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);

//
// basic_string との比較：
//
template <class BidirectionalIterator, class traits, class Allocator>
bool operator == (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator != (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator < (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                         traits,
                                         Allocator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator > (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                         traits,
```

```
                                        Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator >= (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                   const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator <= (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                   const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                   const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                   const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                        traits,
                                        Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                        traits,
                                        Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                   const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                   const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& rhs);

//
// 文字列ポインタとの比較：
//
template <class BidirectionalIterator>
bool operator == (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                   const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator != (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                   const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator < (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator > (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator >= (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                   const sub_match<BidirectionalIterator>& rhs);
```

```
template <class BidirectionalIterator>
bool operator <= (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);

//
// 1 文字との比較：
//
template <class BidirectionalIterator>
bool operator == (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator != (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator < (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator > (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator >= (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator <= (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
```

```
//
// 加算演算子：
//
template <class BidirectionalIterator, class traits, class Allocator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type, traits, Allocator>
   operator + (const std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type,
                                       traits,
                                       Allocator>& s,
               const sub_match<BidirectionalIterator>& m);
template <class BidirectionalIterator, class traits, class Allocator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type, traits, Allocator>
   operator + (const sub_match<BidirectionalIterator>& m,
               const std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type,
                                       traits,
                                       Allocator>& s);
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (typename iterator_traits<BidirectionalIterator>::value_type const* s,
               const sub_match<BidirectionalIterator>& m);
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (const sub_match<BidirectionalIterator>& m,
               typename iterator_traits<BidirectionalIterator>::value_type const * s);
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (typename iterator_traits<BidirectionalIterator>::value_type const& s,
               const sub_match<BidirectionalIterator>& m);
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (const sub_match<BidirectionalIterator>& m,
               typename iterator_traits<BidirectionalIterator>::value_type const& s);
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (const sub_match<BidirectionalIterator>& m1,
               const sub_match<BidirectionalIterator>& m2);

//
// ストリーム挿入子：
//
template <class charT, class traits, class BidirectionalIterator>
basic_ostream<charT, traits>&
   operator << (basic_ostream<charT, traits>& os,
                const sub_match<BidirectionalIterator>& m);

} // namespace boost
```

説明

メンバ

```
typedef typename std::iterator_traits<iterator>::value_type value_type;
```

イテレータが指す型。

```
typedef typename std::iterator_traits<iterator>::difference_type difference_type;
```

2 つのイテレータの差を表す型。

```
typedef BidirectionalIterator iterator;
```

イテレータ型。

```
iterator first
```

マッチの先頭位置を示すイテレータ。

```
iterator second
```

マッチの終端位置を示すイテレータ。

```
bool matched
```

この部分式がマッチしているかを示す論理値。

```
static difference_type length();
```

効果: マッチした部分式の長さを返す。この部分式がマッチしなかった場合は 0 を返す。`matched ? distance(first, second) : 0` と同じ。

```
operator basic_string<value_type>()const;
```

効果: `*this` を文字列に変換する。`(matched ? basic_string<value_type>(first, second) : basic_string<value_type>())` を返す。

```
basic_string<value_type> str()const;
```

効果: `*this` の文字列表現を返す。`(matched ? basic_string<value_type>(first, second) : basic_string<value_type>())` と同じ。

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

効果: `*this` と *s* と字句的比較を行う。`str().compare(s.str())` を返す。

```
int compare(const basic_string<value_type>& s)const;
```

効果: `*this` と文字列 *s* を比較する。`str().compare(s)` を返す。

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

効果: *this と null 終端文字列 *s* を比較する。str().compare(s)を返す。

```
typedef implementation-private capture_sequence_type;
```

効果: 標準ライブラリ Sequence の要件（21.1.1 および表 68 の操作）を満たす実装固有の型を定義する。その value_type は sub_match<BidirectionalIterator>である。この型が std::vector<sub_match<BidirectionalIterator> >となる可能性もあるが、それに依存すべきではない。

```
const capture_sequence_type& captures()const;
```

効果: この部分式に対するすべての捕捉を格納したシーケンスを返す。

事前条件: BOOST_REGEX_MATCH_EXTRA を使ってライブラリをビルドしていなければ、このメンバ関数は定義されない。また正規表現マッチ関数（regex_match、regex_search、regex_iterator、regex_token_iterator）にフラグ match_extra を渡していなければ、有用な情報を返さない。

根拠: この機能を有効にするといくつか影響がある。

- sub_match がより多くのメモリを占有し、複雑な正規表現をマッチする場合にすぐにメモリやスタック空間の不足に陥る。
- match_extra を使用しない場合であっても、処理する機能（例えば独立部分式）によってはマッチアルゴリズムの効率が落ちる。
- match_extra を使用するとさらに効率が落ちる（速度が低下する）。ほとんどの場合、さらに必要なメモリ割り当てが起こる。

sub_match 非メンバ演算子

```
template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

効果: lhs.compare(rhs) == 0 を返す。

```
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

効果: lhs.compare(rhs) != 0 を返す。

```
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

効果: `lhs.compare(rhs) < 0` を返す。

```
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

効果: `lhs.compare(rhs) <= 0` を返す。

```
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

効果: `lhs.compare(rhs) >= 0` を返す。

```
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

効果: `lhs.compare(rhs) > 0` を返す。

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator == (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

効果: `lhs == rhs.str()` を返す。

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator != (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

効果: `lhs != rhs.str()` を返す。

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator < (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                         traits,
                                         Allocator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

効果: `lhs < rhs.str()` を返す。

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator > (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                         traits,
                                         Allocator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

効果: `lhs > rhs.str()` を返す。

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator <= (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

効果: `lhs <= rhs.str()` を返す。

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator >= (const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

効果: `lhs >= rhs.str()` を返す。

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& rhs);
```

効果: `lhs.str() == rhs` を返す。

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& rhs);
```

効果: `lhs.str() != rhs` を返す。

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                         traits,
                                         Allocator>& rhs);
```

効果: `lhs.str() < rhs` を返す。

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                         traits,
                                         Allocator>& rhs);
```

効果: `lhs.str() > rhs` を返す。

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& rhs);
```

効果:lhs.str() <= rhsを返す。

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<iterator_traits<BidirectionalIterator>::value_type,
                                          traits,
                                          Allocator>& rhs);
```

効果:lhs.str() >= rhsを返す。

```
template <class BidirectionalIterator>
bool operator == (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

効果:lhs == rhs.str()を返す。

```
template <class BidirectionalIterator>
bool operator != (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

効果:lhs != rhs.str()を返す。

```
template <class BidirectionalIterator>
bool operator < (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

効果:lhs < rhs.str()を返す。

```
template <class BidirectionalIterator>
bool operator > (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

効果:lhs > rhs.str()を返す。

```
template <class BidirectionalIterator>
bool operator >= (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

効果:lhs >= rhs.str()を返す。

```
template <class BidirectionalIterator>
bool operator <= (typename iterator_traits<BidirectionalIterator>::value_type const* lhs,
```

```
                const sub_match<BidirectionalIterator>& rhs);
```

効果:`lhs <= rhs.str()`を返す。

```
template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
```

効果:`lhs.str() == rhs`を返す。

```
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
```

効果:`lhs.str() != rhs`を返す。

```
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
```

効果:`lhs.str() < rhs`を返す。

```
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
```

効果:`lhs.str() > rhs`を返す。

```
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
```

効果:`lhs.str() >= rhs`を返す。

```
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const* rhs);
```

効果:`lhs.str() <= rhs`を返す。

```
template <class BidirectionalIterator>
bool operator == (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

効果:`lhs == rhs.str()`を返す。

```
template <class BidirectionalIterator>
bool operator != (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

効果:`lhs != rhs.str()`を返す。

```
template <class BidirectionalIterator>
bool operator < (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

効果:`lhs < rhs.str()`を返す。

```
template <class BidirectionalIterator>
bool operator > (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

効果:`lhs > rhs.str()`を返す。

```
template <class BidirectionalIterator>
bool operator >= (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

効果:`lhs >= rhs.str()`を返す。

```
template <class BidirectionalIterator>
bool operator <= (typename iterator_traits<BidirectionalIterator>::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

効果:`lhs <= rhs.str()`を返す。

```
template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
```

効果:`lhs.str() == rhs`を返す。

```
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
```

効果:`lhs.str() != rhs`を返す。

```
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
```

効果:`lhs.str() < rhs`を返す。

```
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
```

効果:lhs.str() > rhs を返す。

```
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
```

効果:lhs.str() >= rhs を返す。

```
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const& rhs);
```

効果:lhs.str() <= rhs を返す。

sub_match の加算演算子により、basic_string に追加可能な型に対して sub_match を追加することができ、結果として新しい文字列を得る。

```
template <class BidirectionalIterator, class traits, class Allocator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type, traits, Allocator>
   operator + (const std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type,
                                       traits,
                                       Allocator>& s,
               const sub_match<BidirectionalIterator>& m);
```

効果:s + m.str()を返す。

```
template <class BidirectionalIterator, class traits, class Allocator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type, traits, Allocator>
   operator + (const sub_match<BidirectionalIterator>& m,
               const std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type,
                                       traits,
                                       Allocator>& s);
```

効果:m.str() + s を返す。

```
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (typename iterator_traits<BidirectionalIterator>::value_type const* s,
               const sub_match<BidirectionalIterator>& m);
```

効果:s + m.str()を返す。

```
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (const sub_match<BidirectionalIterator>& m,
```

```
            typename iterator_traits<BidirectionalIterator>::value_type const* s);
```

効果:m.str() + sを返す。

```
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (typename iterator_traits<BidirectionalIterator>::value_type const& s,
               const sub_match<BidirectionalIterator>& m);
```

効果:s + m.str()を返す。

```
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (const sub_match<BidirectionalIterator>& m,
               typename iterator_traits<BidirectionalIterator>::value_type const& s);
```

効果:m.str() + sを返す。

```
template <class BidirectionalIterator>
std::basic_string<typename iterator_traits<BidirectionalIterator>::value_type>
   operator + (const sub_match<BidirectionalIterator>& m1,
               const sub_match<BidirectionalIterator>& m2);
```

効果:m1.str() + m2.str()を返す。

ストリーム挿入子

```
template <class charT, class traits, class BidirectionalIterator>
basic_ostream<charT, traits>&
   operator << (basic_ostream<charT, traits>& os,
                const sub_match<BidirectionalIterator>& m);
```

効果:(os << m.str())を返す。

regex_match

```
#include <boost/regex.hpp>
```

アルゴリズム regex_match は、与えられた正規表現が双方向イテレータの組で示された文字シーケンス**全体**にマッチするか判定する。このアルゴリズムの定義は以下に示すとおりである。この関数の主な用途は入力データの検証である。

重要

結果が真となるのは式が入力シーケンス**全体**にマッチする場合のみということに注意していただきたい。シーケンス内で式を検索

するには regex_search を使用する。文字列の先頭でマッチを行う場合は、フラグ match_continuous を設定して regex_search を使用する。

```
template <class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 match_results<BidirectionalIterator, Allocator>& m,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);

template <class BidirectionalIterator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);

template <class charT, class Allocator, class traits>
bool regex_match(const charT* str, match_results<const charT*, Allocator>& m,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);

template <class ST, class SA, class Allocator, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 match_results<typename basic_string<charT, ST, SA>::const_iterator, Allocator>& m,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);

template <class charT, class traits>
bool regex_match(const charT* str,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);

template <class ST, class SA, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);
```

説明

```
template <class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 match_results<BidirectionalIterator, Allocator>& m,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);
```

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

効果: 正規表現 *e* と文字シーケンス [first, last) 全体の間に完全なマッチが存在するか判定する。引数 *flags*（match_flag_type を見よ）は、正規表現が文字シーケンスに対してどのようにマッチするかを制御するのに使用する。完全なマッチが存在する場合は真を、それ以外の場合は偽を返す。

例外: 長さ N の文字列に対して式のマッチの計算量が $O(N^2)$ を超え始めた場合、正規表現のマッチ中にプログラムのスタック空間が枯渇した場合（Boost.Regex が再帰モードを使うように構成されているとき）、あるいはマッチオブジェクトが許可されているメモリ

割り当てを消耗しきった場合（Boost.Regex が非再帰モードを使うように構成されているとき）に std::runtime_error。

事後条件: 関数が偽を返した場合、引数 *m* の状態は未定義である。それ以外の場合は次の表のとおりである。

要素	値
m.size()	1 + e.mark_count()
m.empty()	false
m.prefix().first	first
m.prefix().last	first
m.prefix().matched	false
m.suffix().first	last
m.suffix().last	last
m.suffix().matched	false
m[0].first	first
m[0].second	last
m[0].matched	完全マッチが見つかった場合は真、（match_partial フラグを設定した結果）部分マッチが見つかった場合は偽。
m[n].first	n < m.size() であるすべての整数について部分式 *n* にマッチしたシーケンスの先頭。それ以外で部分式 *n* がマッチしなかった場合は last。
m[n].second	n < m.size() であるすべての整数について部分式 *n* にマッチしたシーケンスの終端。それ以外で部分式 *n* がマッチしなかった場合は last。
m[n].matched	n < m.size() であるすべての整数について部分式 *n* がマッチした場合は真、それ以外は偽。

```
template <class BidirectionalIterator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);
```

効果: match_results<BidirectionalIterator>のインスタンス what を構築し、regex_match(first, last, what, e, flags)の結果を返す。

```
template <class charT, class Allocator, class traits>
bool regex_match(const charT* str, match_results<const charT*, Allocator>& m,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);
```

効果: regex_match(str, str + char_traits<charT>::length(str), m, e, flags)の結果を返す。

```
template <class ST, class SA, class Allocator,
          class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 match_results<typename basic_string<charT, ST, SA>::const_iterator, Allocator>& m,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);
```

効果:regex_match(s.begin(), s.end(), m, e, flags)の結果を返す。

```
template <class charT, class traits>
bool regex_match(const charT* str,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);
```

効果:regex_match(str, str + char_traits<charT>::length(str), e, flags)を返す。

```
template <class ST, class SA, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 const basic_regex <charT, traits>& e,
                 match_flag_type flags = match_default);
```

効果:regex_match(s.begin(), s.end(), e, flags)を返す。

使用例

以下はFTP応答を処理する例である。

```
#include <stdlib.h>
#include <boost/regex.hpp>
#include <string>
#include <iostream>

using namespace boost;

regex expression("([0-9]+)(\\-| |$)(.*)");

// process_ftp：
// 成功時はFTP応答コードを返し、
// msgに応答メッセージを書き込む。
int process_ftp(const char* response, std::string* msg)
{
   cmatch what;
   if(regex_match(response, what, expression))
   {
      // what[0]には文字列全体が入る
      // what[1]には応答コードが入る
      // what[2]には区切り文字が入る
      // what[3]にはテキストメッセージが入る。
      if(msg)
         msg->assign(what[3].first, what[3].second);
      return std::atoi(what[1].first);
   }
   // マッチしなかったら失敗
   if(msg)
      msg->erase();
   return -1;
}
```

regex_search

```
#include <boost/regex.hpp>
```

アルゴリズム regex_search は、双方向イテレータの組で示される範囲から与えられた正規表現を検索する。このアルゴリズムは様々な発見的方法を用いて検索時間を短縮する。そのために、個々の位置からマッチが開始する可能性があるかチェックのみを行う。このアルゴリズムの定義は以下のとおりである。

```
template <class BidirectionalIterator,
          class Allocator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                  match_results<BidirectionalIterator, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);

template <class ST, class SA,
          class Allocator, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
                  match_results<
                     typename basic_string<charT, ST,SA>::const_iterator,
                     Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);

template<class charT, class Allocator, class traits>
bool regex_search(const charT* str,
                  match_results<const charT*, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);

template <class BidirectionalIterator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);

template <class charT, class traits>
bool regex_search(const charT* str,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);

template<class ST, class SA, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);
```

説明

```
template <class BidirectionalIterator,
          class Allocator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                  match_results<BidirectionalIterator, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);
```

要件: 型 `BidirectionalIterator` が双方向イテレータの要件（24.1.4）を満たす。

効果: [first, last) 中に正規表現 *e* にマッチする部分シーケンスが存在するか判定する。引数 *flags* は、式が文字シーケンスに対してどのようにマッチするかを制御するのに使用する。完全なマッチが存在する場合は真を、それ以外の場合は偽を返す。

例外: 長さ *N* の文字列に対して式のマッチの計算量が $O(N^2)$を超え始めた場合、式のマッチ中にプログラムのスタック空間が枯渇した場合（Boost.Regex が再帰モードを使うように構成されているとき）、あるいはマッチオブジェクトが許可されているメモリ割り当てを消耗しきった場合（Boost.Regex が非再帰モードを使うように構成されているとき）に `std::runtime_error`。

事後条件: 関数が偽を返した場合、引数 *m* の状態は未定義である。それ以外の場合は次の表のとおりである。

要素	値
`m.size()`	`1 + e.mark_count()`
`m.empty()`	`false`
`m.prefix().first`	`first`
`m.prefix().last`	`m[0].first`
`m.prefix().matched`	`m.prefix().first != m.prefix.second`
`m.suffix().first`	`m[0].second`
`m.suffix().last`	`last`
`m.suffix().matched`	`m.suffix().first != m.suffix().second`
`m[0].first`	正規表現にマッチした文字シーケンスの先頭
`m[0].second`	正規表現にマッチした文字シーケンスの終端
`m[0].matched`	完全マッチが見つかった場合は真、（`match_partial` フラグを設定した結果）部分マッチが見つかった場合は偽。
`m[n].first`	`n < m.size()` であるすべての整数について部分式 *n* にマッチしたシーケンスの先頭。それ以外で部分式 *n* がマッチしなかった場合は `last`。
`m[n].second`	`n < m.size()` であるすべての整数について部分式 *n* にマッチしたシーケンスの終端。それ以外で部分式 *n* がマッチしなかった場合は `last`。
`m[n].matched`	`n < m.size()` であるすべての整数について部分式 *n* がマッチした場合は真、それ以外は偽。

```
template<class charT, class Allocator, class traits>
bool regex_search(const charT* str,
                  match_results<const charT*, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);
```

効果: `regex_search(str, str + char_traits<charT>::length(str), m, e, flags)` の結果を返す。

```
template <class ST, class SA, Allocator, class charT,
          class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
                  match_results<typename basic_string<charT, ST,SA>::const_iterator, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);
```

効果: `regex_search(s.begin(), s.end(), m, e, flags)` の結果を返す。

```
template <class BidirectionalIterator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);
```

効果: match_results<BidirectionalIterator>のインスタンス what を構築し、regex_search(first, last, what, e, flags)の結果を返す。

```
template <class charT, class traits>
bool regex_search(const charT* str,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);
```

効果: regex_search(str, str + char_traits<charT>::length(str), e, flags)の結果を返す。

```
template<class ST, class SA, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits>& e,
                  match_flag_type flags = match_default);
```

効果: regex_search(s.begin(), s.end(), e, flags)の結果を返す。

使用例

以下の例は、ファイルの内容を 1 つの文字列として読み取り、ファイル内の C++クラス宣言をすべて検索する。このコードは std::string の実装方法に依存しない。例えば SGI の rope クラス（不連続メモリバッファが使われている）を使うように容易に修正できる。

```
#include <string>
#include <map>
#include <boost/regex.hpp>

// 目的：
// ファイルの内容を単一の文字列として受け取り、
// C++クラス宣言をすべて検索し、それらの位置を
// 文字列対整数の辞書に保存する
typedef std::map<std::string, int, std::less<std::string> > map_type;

boost::regex expression(
   "^(template[[:space:]]*<[^;:{]+>[[:space:]]*)?"
   "(class|struct)[[:space:]]*"
   "(\\<\\w+\\>([[:blank:]]*\\([^)]*\\))?"
   "[[:space:]]*)*(\\<\\w*\\>)[[:space:]]*"
   "(<[^;:{]+>[[:space:]]*)?(\\{|:[^;\\{()]*\\{)");

void IndexClasses(map_type& m, const std::string& file)
{
   std::string::const_iterator start, end;
   start = file.begin();
   end = file.end();
   boost::match_results<std::string::const_iterator> what;
```

```
   boost::match_flag_type flags = boost::match_default;
   while(regex_search(start, end, what, expression, flags))
   {
      // what[0]には文字列全体が入り
      // what[5]にはクラス名が入る。
      // what[6]にはテンプレートの特殊化（あれば）が入り、
      // クラス名と位置を辞書に入れて対応させる：
      m[std::string(what[5].first, what[5].second)
            + std::string(what[6].first, what[6].second)]
         = what[5].first - file.begin();
      // 検索位置を更新する：
      start = what[0].second;
      // flags を更新する：
      flags |= boost::match_prev_avail;
      flags |= boost::match_not_bob;
   }
}
```

regex_replace

```
#include <boost/regex.hpp>
```

アルゴリズム regex_replace は文字列を検索して正規表現に対するマッチをすべて発見する。さらに各マッチについて match_results<>::format を呼び出して文字列を書式化し、結果を出力イテレータに送る。マッチしなかったテキスト部分は *flags* 引数にフラグ format_no_copy が設定されていない場合に限り、変更を加えず出力にコピーする。フラグ format_first_only が設定されている場合は、すべてのマッチではなく最初のマッチのみ置換する。

```
template <class OutputIterator, class BidirectionalIterator, class traits, class Formatter>
OutputIterator regex_replace(OutputIterator out,
                             BidirectionalIterator first,
                             BidirectionalIterator last,
                             const basic_regex<charT, traits>& e,
                             Formatter fmt,
                             match_flag_type flags = match_default);

template <class traits, class Formatter>
basic_string<charT> regex_replace(const basic_string<charT>& s,
                                  const basic_regex<charT, traits>& e,
                                  Formatter fmt,
                                  match_flag_type flags = match_default);
```

説明

```
template <class OutputIterator, class BidirectionalIterator, class traits, class Formatter>
OutputIterator regex_replace(OutputIterator out,
                             BidirectionalIterator first,
                             BidirectionalIterator last,
                             const basic_regex<charT, traits>& e,
```

```
                                Formatter fmt,
                                match_flag_type flags = match_default);
```

シーケンス [first, last) 中の正規表現 *e* に対するすべてのマッチを列挙し、各マッチと書式化文字列 *fmt* をマージして得られる文字列で置換し、結果の文字列を *out* にコピーする。*fmt* が単項・二項・三項関数オブジェクトである場合、関数オブジェクトが生成した文字シーケンスは変更を加えられることなく出力にコピーされる。

flags に `format_no_copy` が設定されている場合、マッチしなかったテキスト部分は出力にコピーされない。

flags に `format_first_only` が設定されている場合、*e* の最初のマッチのみが置換される。

書式化文字列 *fmt* およびマッチ検索に使用する規則は *flags* に設定されているフラグにより決定する。match_flag_type を見よ。

要件：型 `Formatter` は `char_type[]` 型の null 終端文字列へのポインタ、`char_type` 型のコンテナ（例えば `std::basic_string<char_type>`）、あるいは関数呼び出しにより置換文字列を生成する単項・二項・三項関数子のいずれかでなければならない。関数子の場合、`fmt(what)` は置換テキストと使用する `char_type` のコンテナを返さなければならず、`fmt(what, out)` および `fmt(what, out, flags)` はいずれも置換テキストを`*out` に出力し `OutputIterator` の新しい位置を返さなければならない。以上において `what` は見つかったマッチを表す match_results オブジェクトである。書式化オブジェクトが関数子の場合、**値渡しされる**ことに注意していただきたい。関数オブジェクトを内部状態とともに渡す場合は、Boost.Ref を使ってオブジェクトが参照渡しされるようラップするとよい。

効果: regex_iterator オブジェクトを構築し、

```
regex_iterator<BidirectionalIterator, charT, traits, Allocator>
                                        i(first, last, e, flags),
```

i を使ってシーケンス [first, last) 中のすべてのマッチ *m*（match_results<BidirectionalIterator>型）を列挙する。

マッチが見つからず、かつ

```
!(flags & format_no_copy)
```

であれば、次を呼び出す。

```
std::copy(first, last, out)
```

それ以外で

```
!(flags & format_no_copy)
```

であれば、

```
std::copy(m.prefix().first, m.prefix().last, out)
```

を呼び出し、次を呼び出す。

```
m.format(out, fmt, flags)
```

以上のいずれにも該当せず、

```
!(flags & format_no_copy)
```

であれば、次を呼び出す。

```
std::copy(last_m.suffix().first, last_m,suffix().last, out)
```

ただし *last_m* は最後に見つかったマッチのコピーである。

`flags & format_first_only` が非 0 であれば、最初に見つかったマッチのみを置換する。

例外: 長さ *N* の文字列に対して式のマッチの計算量が $O(N^2)$ を超え始めた場合、式のマッチ中にプログラムのスタック空間が枯渇した場合（Boost.Regex が再帰モードを使うように構成されているとき）、あるいはマッチオブジェクトが許可されているメモリ割り当てを消耗しきった場合（Boost.Regex が非再帰モードを使うように構成されているとき）に `std::runtime_error`。

戻り値: *out*。

```
template <class traits, class Formatter>
basic_string<charT> regex_replace(const basic_string<charT>& s,
                                  const basic_regex<charT, traits>& e,
                                  Formatter fmt,
                                  match_flag_type flags = match_default);
```

要件: 型 `Formatter` は `char_type[]` 型の null 終端文字列へのポインタ、`char_type` 型のコンテナ（例えば `std::basic_string<char_type>`）、あるいは関数呼び出しにより置換文字列を生成する単項・二項・三項関数子のいずれかでなければならない。関数子の場合、`fmt(what)` は置換テキストと使用する `char_type` のコンテナを返さなければならず、`fmt(what, out)` および `fmt(what, out, flags)` はいずれも置換テキストを `*out` に出力し `OutputIterator` の新しい位置を返さなければならない。以上において `what` は見つかったマッチを表す `match_results` オブジェクトである。

効果: オブジェクト `basic_string<charT> result` を構築し、`regex_replace(back_inserter(result), s.begin(), s.end(), e, fmt, flags)` を呼び出し、`result` を返す。

使用例

次の例は C/C++ソースコードを入力として受け取り、構文強調した HTML コードを出力する。

```
#include <fstream>
#include <sstream>
#include <string>
#include <iterator>
#include <boost/regex.hpp>
#include <fstream>
#include <iostream>

// 目的：
```

```
// ファイルの内容を受け取り、構文強調した
// HTML 形式に変換する

boost::regex e1, e2;
extern const char* expression_text;
extern const char* format_string;
extern const char* pre_expression;
extern const char* pre_format;
extern const char* header_text;
extern const char* footer_text;

void load_file(std::string& s, std::istream& is)
{
   s.erase();
   s.reserve(is.rdbuf()->in_avail());
   char c;
   while(is.get(c))
   {
      if(s.capacity() == s.size())
         s.reserve(s.capacity() * 3);
      s.append(1, c);
   }
}

int main(int argc, const char** argv)
{
   try{
      e1.assign(expression_text);
      e2.assign(pre_expression);
      for(int i = 1; i < argc; ++i)
      {
         std::cout << "次のファイルを処理中 " << argv[i] << std::endl;
         std::ifstream fs(argv[i]);
         std::string in;
         load_file(in, fs);
         std::string out_name(std::string(argv[i]) + std::string(".htm"));
         std::ofstream os(out_name.c_str());
         os << header_text;
         // 最初に一時文字列ストリームに出力して
         // '<' と '>' を取り去る
         std::ostringstream t(std::ios::out | std::ios::binary);
         std::ostream_iterator<char, char> oi(t);
         boost::regex_replace(oi, in.begin(), in.end(),
            e2, pre_format, boost::match_default | boost::format_all);
         // 次に最終的な出力ストリームに出力し
         // 構文強調を追加する：
         std::string s(t.str());
         std::ostream_iterator<char, char> out(os);
         boost::regex_replace(out, s.begin(), s.end(),
            e1, format_string, boost::match_default | boost::format_all);
         os << footer_text;
      }
   }
   catch(...)
   { return -1; }
   return 0;
}

extern const char* pre_expression = "(<)|(>)|(&)|\\r";
```

```
extern const char* pre_format = "(?1<)(?2>)(?3&amp;)";


const char* expression_text =
   // プリプロセッサディレクティブ：添字１
   "(^[[:blank:]]*#(?:[^\\\\\\n]|\\\\[^\\n[:punct:][:word:]]*[\\n[:punct:][:word:]])*)|"
   // 注釈：添字２
   "(//[^\\n]*|/\\*.*?\\*/)|"
   // 直値：添字３
   "\\<([+-]?(?:(?:0x[[:xdigit:]]+)|(?:(?:[[:digit:]]*\\.)?[[:digit:]]+"
   "(?:[eE][+-]?[[:digit:]]+)?))u?(?:(?:int(?:8|16|32|64))|L)?)\\>|"
   // 文字列直値：添字４
   "('(?:[^\\\\']|\\\\.)*'|\"(?:[^\\\\\"]|\\\\.)*\")|"
   // キーワード：添字５
   "\\<(__asm|__cdecl|__declspec|__export|__far16|__fastcall|__fortran|__import"
   "|__pascal|__rtti|__stdcall|_asm|_cdecl|__except|_export|_far16|_fastcall"
   "|__finally|_fortran|_import|_pascal|_stdcall|__thread|__try|asm|auto|bool"
   "|break|case|catch|cdecl|char|class|const|const_cast|continue|default|delete"
   "|do|double|dynamic_cast|else|enum|explicit|extern|false|float|for|friend|goto"
   "|if|inline|int|long|mutable|namespace|new|operator|pascal|private|protected"
   "|public|register|reinterpret_cast|return|short|signed|sizeof|static|static_cast"
   "|struct|switch|template|this|throw|true|try|typedef|typeid|typename|union|unsigned"
   "|using|virtual|void|volatile|wchar_t|while)\\>"
   ;

const char* format_string = "(?1<font color=\"#008040\">$&</font>)"
                            "(?2<I><font color=\"#000080\">$&</font></I>)"
                            "(?3<font color=\"#0000A0\">$&</font>)"
                            "(?4<font color=\"#0000FF\">$&</font>)"
                            "(?5<B>$&</B>)";

const char* header_text =
   "<HTML>\n<HEAD>\n"
   "<TITLE>Auto-generated html formated source</TITLE>\n"
   "<META HTTP-EQUIV=\"Content-Type\Type\" CONTENT=\"text/html; charset=windows-1252\">\n"
   "</HEAD>\n"
   "<BODY LINK=\"#0000ff\" VLINK=\"#800080\" BGCOLOR=\"#ffffff\">\n"
   "<P> </P>\n<PRE>";

const char* footer_text = "</PRE>\n</BODY>\n\n";
```

regex_iterator

イテレータ型 regex_iterator はシーケンス中で見つかった正規表現マッチをすべて列挙する。regex_iterator を逆参照すると match_results オブジェクトへの参照が得られる。

```
template <class BidirectionalIterator,
          class charT = iterator_traits<BidirectionalIterator>::value_type,
          class traits = regex_traits<charT> >
class regex_iterator
{
public:
   typedef          basic_regex<charT, traits>                                regex_type;
   typedef          match_results<BidirectionalIterator>                      value_type;
   typedef typename iterator_traits<BidirectionalIterator>::difference_type difference_type;
```

```
   typedef          const value_type*                                    pointer;
   typedef          const value_type&                                    reference;
   typedef          std::forward_iterator_tag                            iterator_category;

   regex_iterator();
   regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
                  const regex_type& re,
                  match_flag_type m = match_default);
   regex_iterator(const regex_iterator&);
   regex_iterator& operator=(const regex_iterator&);
   bool operator==(const regex_iterator&)const;
   bool operator!=(const regex_iterator&)const;
   const value_type& operator*()const;
   const value_type* operator->()const;
   regex_iterator& operator++();
   regex_iterator operator++(int);
};

typedef regex_iterator<const char*>                  cregex_iterator;
typedef regex_iterator<std::string::const_iterator>  sregex_iterator;

#ifndef BOOST_NO_WREGEX
typedef regex_iterator<const wchar_t*>               wcregex_iterator;
typedef regex_iterator<std::wstring::const_iterator> wsregex_iterator;
#endif

template <class charT, class traits> regex_iterator<const charT*, charT, traits>
   make_regex_iterator(const charT* p, const basic_regex<charT, traits>& e,
                       regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, class ST, class SA>
regex_iterator<typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
   make_regex_iterator(const std::basic_string<charT, ST, SA>& p,
                       const basic_regex<charT, traits>& e,
                       regex_constants::match_flag_type m = regex_constants::match_default);
```

説明

regex_iterator はイテレータの組で構築され、イテレータ範囲の正規表現マッチをすべて列挙する。

```
regex_iterator();
```

効果: シーケンスの終了を指す regex_iterator を構築する。

```
regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
               const regex_type& re,
               match_flag_type m = match_default);
```

効果: シーケンス [a,b) 内で正規表現 *re* と match_flag_type *m* を使って見つかるすべてのマッチを列挙する regex_iterator を構築する。オブジェクト *re* は regex_iterator の生涯にわたって存在していなければならない。

例外: 長さ *N* の文字列に対して式のマッチの計算量が $O(N^2)$を超え始めた場合、式のマッチ中にプログラムのスタック空間が枯渇した場合（Boost.Regex が再帰モードを使うように構成されているとき）、あるいはマッチオブジェクトが許可されているメモリ割り当てを消耗しきった場合（Boost.Regex が非再帰モードを使うように構成されているとき）に std::runtime_error。

```
regex_iterator(const regex_iterator& that);
```

効果: that のコピーを構築する。

事後条件: *this == that。

```
regex_iterator& operator=(const regex_iterator& that);
```

効果: *this を that と等価にする。

事後条件: *this == that。

```
bool operator==(const regex_iterator& that)const;
```

効果: *this と that が等価であれば真を返す。

```
bool operator!=(const regex_iterator& that)const;
```

効果: !(*this == that)を返す。

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

効果: regex_iterator の逆参照は match_results オブジェクトへの参照である。そのメンバは次のとおりである。

要素	値
(*it).size()	1 + re.mark_count()
(*it).empty()	false
(*it).prefix().first	最後に見つかったマッチの終端。最初の列挙の場合は対象シーケンスの先頭。
(*it).prefix().last	見つかったマッチの先頭と同じ。(*it)[0].first
(*it).prefix().matched	マッチ全体より前の部分が空文字列でなければ真。(*it).prefix().first != (*it).prefix().second
(*it).suffix().first	見つかったマッチの終端と同じ。(*it)[0].second
(*it).suffix().last	対象シーケンスの終端。
(*it).suffix().matched	マッチ全体より後ろの部分が空文字列でなければ真。(*it).suffix().first != (*it).suffix().second
(*it)[0].first	正規表現にマッチした文字シーケンスの先頭。
(*it)[0].second	正規表現にマッチした文字シーケンスの終端。
(*it)[0].matched	完全マッチが見つかった場合は真、(match_partial フラグを設定した結果)部分マッチが見つかった場合は偽。
(*it)[n].first	n < (*it).size() であるすべての整数について部分式 *n* にマッチしたシーケンスの先頭。それ以外で部分式 *n* がマッチしなかった場合は last。
(*it)[n].second	n < (*it).size() であるすべての整数について部分式 *n* にマッチしたシーケンスの終端。そ

要素	値
	れ以外で部分式 *n* がマッチしなかった場合は `last`。
`(*it)[n].matched`	`n < (*it).size()` であるすべての整数について部分式 *n* がマッチした場合は真、それ以外は偽。
`(*it).position(n)`	`n < (*it).size()` であるすべての整数について、対象シーケンスの先頭から部分式 *n* の先頭までの距離。

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

効果: `&(*this)` を返す。

```
regex_iterator& operator++();
```

効果: イテレータを対象シーケンス中の次のマッチに移動する。何も見つからない場合はシーケンスの終端に移動する。最後のマッチが長さ 0 の文字列へのマッチである場合は、regex_iterator は以下の要領で次のマッチを検索する。非 0 長のマッチが最後のマッチと同じ位置から始まっている場合は、そのマッチを返す。それ以外の場合は次のマッチ（再び長さが 0 ということもありうる）を最後のマッチの 1 つ右の位置から検索する。

例外: 長さ N の文字列に対して式のマッチの計算量が $O(N^2)$ を超え始めた場合、式のマッチ中にプログラムのスタック空間が枯渇した場合（Boost.Regex が再帰モードを使うように構成されているとき）、あるいはマッチオブジェクトが許可されているメモリ割り当てを消耗しきった場合（Boost.Regex が非再帰モードを使うように構成されているとき）に `std::runtime_error`。

戻り値: `*this`。

```
regex_iterator operator++(int);
```

効果: 戻り値用に `*this` のコピーを構築した後、`++(*this)` を呼び出す。

戻り値: 結果。

```
template <class charT, class traits>
regex_iterator<const charT*, charT, traits>
   make_regex_iterator(const charT* p, const basic_regex<charT, traits>& e,
                       regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, class ST, class SA>
regex_iterator<typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
   make_regex_iterator(const std::basic_string<charT, ST, SA>& p,
                       const basic_regex<charT, traits>& e,
                       regex_constants::match_flag_type m = regex_constants::match_default);
```

戻り値: 式 *e* と match_flag_type *m* を用いてテキスト *p* 中で見つかるすべてのマッチを列挙するイテレータを返す。

使用例

次の例は C++ソースファイルを受け取り、クラス名とそのクラスのファイル内での位置を含んだ索引を作成する。

```
#include <string>
#include <map>
#include <fstream>
#include <iostream>
#include <boost/regex.hpp>

using namespace std;

// 目的：
// ファイルの内容を１つの文字列として受け取り
// C++クラス定義をすべて検索し、それらの位置を
// 文字列対整数の辞書に保存する。

typedef std::map<std::string, std::string::difference_type, std::less<std::string> > map_type;

const char* re =
   // 前に空白があってもよい：
   "^[[:space:]]*"
   // テンプレート宣言があってもよい：
   "(template[[:space:]]*<[^;:{]+>[[:space:]]*)?"
   // classかstruct：
   "(class|struct)[[:space:]]*"
   // declspecマクロなど：
   "("
      "\\<\\w+\\>"
      "("
         "[[:blank:]]*\\([^)]*\\)"
      ")?"
      "[[:space:]]*"
   ")*"
   // クラス名
   "(\\<\\w*\\>)[[:space:]]*"
   // テンプレート特殊化引数
   "(<[^;:{]+>)?[[:space:]]*"
   // {か:で終了
   "(\\{|:[^;\\{()]*\\{)";

boost::regex expression(re);
map_type class_index;

bool regex_callback(const boost::match_results<std::string::const_iterator>& what)
{
   // what[0]には文字列全体が入り
   // what[5]にはクラス名が入る。
   // what[6]にはテンプレートの特殊化が入る（あれば）。
   // クラス名と位置を辞書に入れる：
   class_index[what[5].str() + what[6].str()] = what.position(5);
   return true;
}

void load_file(std::string& s, std::istream& is)
{
   s.erase();
   s.reserve(is.rdbuf()->in_avail());
   char c;
   while(is.get(c))
   {
```

```
      if(s.capacity() == s.size())
         s.reserve(s.capacity() * 3);
      s.append(1, c);
   }
}

int main(int argc, const char** argv)
{
   std::string text;
   for(int i = 1; i < argc; ++i)
   {
      cout << "次のファイルを処理中 " << argv[i] << endl;
      std::ifstream fs(argv[i]);
      load_file(text, fs);
      // イテレータを構築しておく：
      boost::sregex_iterator m1(text.begin(), text.end(), expression);
      boost::sregex_iterator m2;
      std::for_each(m1, m2, &regex_callback);
      // 結果をコピーする：
      cout << class_index.size() << " 個のマッチが見つかりました" << endl;
      map_type::iterator c, d;
      c = class_index.begin();
      d = class_index.end();
      while(c != d)
      {
         cout << "クラス \"" << (*c).first << "\" が次の位置で見つかりました：" << (*c).second << endl;
         ++c;
      }
      class_index.erase(class_index.begin(), class_index.end());
   }
   return 0;
}
```

regex_token_iterator

テンプレートクラス regex_token_iterator はイテレータアダプタである。すなわち、入力テキストシーケンス内の正規表現マッチをすべて検索することで既存のシーケンス（入力テキスト）に対する新しいビューを表現し、各マッチ文字シーケンスを与える。このイテレータが列挙する各位置は、正規表現中の各部分式のマッチを表す sub_match オブジェクトである。regex_token_iterator クラスを使用して-1 の添字で部分式を列挙すると、イテレータはフィールド分割を行う。すなわち、指定した正規表現にマッチしない各文字コンテナシーケンスにつき 1 つの文字シーケンスを列挙する。[6]

```
template <class BidirectionalIterator,
          class charT = iterator_traits<BidirectionalIterator>::value_type,
          class traits = regex_traits<charT> >
class regex_token_iterator
{
public:
   typedef          basic_regex<charT, traits>                                regex_type;
   typedef          sub_match<BidirectionalIterator>                          value_type;
   typedef typename iterator_traits<BidirectionalIterator>::difference_type difference_type;
```

6 訳注 -1 の添字は、後述するように実際には sub_match と同様に「前回のマッチの終端から今回のマッチの先頭まで」を表します。sub_match の項でドキュメントされていない-2 の添字についても同様ですが、奇妙な動作をするので使用しないほうが無難です。

```
   typedef          const value_type*                                    pointer;
   typedef          const value_type&                                    reference;
   typedef          std::forward_iterator_tag                            iterator_category;

   regex_token_iterator();
   regex_token_iterator(BidirectionalIterator a,
                        BidirectionalIterator b,
                        const regex_type& re,
                        int submatch = 0,
                        match_flag_type m = match_default);
   regex_token_iterator(BidirectionalIterator a,
                        BidirectionalIterator b,
                        const regex_type& re,
                        const std::vector<int>& submatches,
                        match_flag_type m = match_default);
   template <std::size_t N>
   regex_token_iterator(BidirectionalIterator a,
                        BidirectionalIterator b,
                        const regex_type& re,
                        const int (&submatches)[N],
                        match_flag_type m = match_default);
   regex_token_iterator(const regex_token_iterator&);
   regex_token_iterator& operator=(const regex_token_iterator&);
   bool operator==(const regex_token_iterator&)const;
   bool operator!=(const regex_token_iterator&)const;
   const value_type& operator*()const;
   const value_type* operator->()const;
   regex_token_iterator& operator++();
   regex_token_iterator operator++(int);
};

typedef regex_token_iterator<const char*> cregex_token_iterator;
typedef regex_token_iterator<std::string::const_iterator> sregex_token_iterator;
#ifndef BOOST_NO_WREGEX
typedef regex_token_iterator<const wchar_t*> wcregex_token_iterator;
typedef regex_token_iterator<std::wstring::const_iterator> wsregex_token_iterator;
#endif

template <class charT, class traits>
regex_token_iterator<const charT*, charT, traits>
   make_regex_token_iterator(
         const charT* p,
         const basic_regex<charT, traits>& e,
         int submatch = 0,
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, class ST, class SA>
regex_token_iterator<typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
   make_regex_token_iterator(
         const std::basic_string<charT, ST, SA>& p,
         const basic_regex<charT, traits>& e,
         int submatch = 0,
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, std::size_t N>
regex_token_iterator<const charT*, charT, traits>
   make_regex_token_iterator(
         const charT* p,
         const basic_regex<charT, traits>& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);
```

```
template <class charT, class traits, class ST, class SA, std::size_t N>
regex_token_iterator<typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
   make_regex_token_iterator(
         const std::basic_string<charT, ST, SA>& p,
         const basic_regex<charT, traits>& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits>
regex_token_iterator<const charT*, charT, traits>
   make_regex_token_iterator(
         const charT* p,
         const basic_regex<charT, traits>& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, class ST, class SA>
regex_token_iterator<
      typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
   make_regex_token_iterator(
         const std::basic_string<charT, ST, SA>& p,
         const basic_regex<charT, traits>& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m = regex_constants::match_default);
```

説明

```
regex_token_iterator();
```

効果: シーケンスの終端を指すイテレータを構築する。

```
regex_token_iterator(BidirectionalIterator a,
                     BidirectionalIterator b,
                     const regex_type& re,
                     int submatch = 0,
                     match_flag_type m = match_default);
```

事前条件: `!re.empty()`。オブジェクト *re* はイテレータの生涯にわたって存在しなければならない。

効果: シーケンス [a,b) 中で、式 *re* とマッチフラグ *m*（match_flag_type を見よ）で見つかる各正規表現マッチに対して文字列を 1 つずつ列挙する regex_token_iterator を構築する。列挙される文字列は、見つかった各マッチに対する部分式 *submatch* である。*submatch* が-1 の場合は、式 *re* にマッチしなかったテキストシーケンスをすべて列挙する（フィールドの分割）。

例外: 長さ N の文字列に対して式のマッチの計算量が $O(N^2)$を超え始めた場合、式のマッチ中にプログラムのスタック空間が枯渇した場合（Boost.Regex が再帰モードを使うように構成されているとき）、あるいはマッチオブジェクトが許可されているメモリ割り当てを消耗しきった場合（Boost.Regex が非再帰モードを使うように構成されているとき）に `std::runtime_error`。

```
regex_token_iterator(BidirectionalIterator a,
                     BidirectionalIterator b,
                     const regex_type& re,
                     const std::vector<int>& submatches,
```

```
                    match_flag_type m = match_default);
```

事前条件: `submatches.size() && !re.empty()`。オブジェクト *re* はイテレータの生涯にわたって存在しなければならない。

効果: シーケンス [a,b) 中で、式 *re* とマッチフラグ *m*（match_flag_type を見よ）で見つかる各正規表現マッチに対して `submatches.size()` 個の文字列を列挙する regex_token_iterator を構築する。各マッチに対して、ベクタ *submatches* 内の添字に対応する各部分式にマッチした文字列を 1 つずつ列挙する。`submatches[0]` が-1 の場合、各マッチに対して最初に列挙する文字列は、前回のマッチの終端から今回のマッチの先頭までのテキストとなり、さらにこれ以上マッチが見つからない場合に列挙する文字列（最後のマッチの終端から対象シーケンスの終端までのテキスト）が 1 つ追加される。

例外: 長さ *N* の文字列に対して式のマッチの計算量が $O(N^2)$ を超え始めた場合、式のマッチ中にプログラムのスタック空間が枯渇した場合（Boost.Regex が再帰モードを使うように構成されているとき）、あるいはマッチオブジェクトが許可されているメモリ割り当てを消耗しきった場合（Boost.Regex が非再帰モードを使うように構成されているとき）に `std::runtime_error`。

```
template <std::size_t R>
regex_token_iterator(BidirectionalIterator a,
                     BidirectionalIterator b,
                     const regex_type& re,
                     const int (&submatches)[R],
                     match_flag_type m = match_default);
```

事前条件: `!re.empty()`。オブジェクト *re* はイテレータの生涯にわたって存在しなければならない。

効果: シーケンス [a,b) 中で、式 *re* とマッチフラグ *m*（match_flag_type を見よ）で見つかる各正規表現マッチに対して *R* 個の文字列を列挙する regex_token_iterator を構築する。各マッチに対して、配列 *submatches* 内の添字に対応する各部分式にマッチした文字列を 1 つずつ列挙する。`submatches[0]` が-1 の場合、各マッチに対して最初に列挙する文字列は、前回のマッチの終端から今回のマッチの先頭までのテキストとなり、さらにこれ以上マッチが見つからない場合に列挙する文字列（最後のマッチの終端から対象シーケンスの終端までのテキスト）が 1 つ追加される。

例外: 長さ *N* の文字列に対して式のマッチの計算量が $O(N^2)$ を超え始めた場合、式のマッチ中にプログラムのスタック空間が枯渇した場合（Boost.Regex が再帰モードを使うように構成されているとき）、あるいはマッチオブジェクトが許可されているメモリ割り当てを消耗しきった場合（Boost.Regex が非再帰モードを使うように構成されているとき）に `std::runtime_error`。

```
regex_token_iterator(const regex_token_iterator& that);
```

効果: `that` のコピーを構築する。

事後条件: `*this == that`。

```
regex_token_iterator& operator=(const regex_token_iterator& that);
```

効果: `*this` を `that` と等価にする。

事後条件: `*this == that`。

```
bool operator==(const regex_token_iterator& that)const;
```

効果: *this と that が同じ位置であれば真を返す。

```
bool operator!=(const regex_token_iterator& that)const;
```

効果: !(*this == that)を返す。

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

効果: 列挙中の現在の文字シーケンスを返す。

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

効果: &(*this)を返す。

```
regex_token_iterator& operator++();
```

効果: 列挙中の次の文字シーケンスへ移動する。

例外: 長さ N の文字列に対して式のマッチの計算量が $O(N^2)$ を超え始めた場合、式のマッチ中にプログラムのスタック空間が枯渇した場合（Boost.Regex が再帰モードを使うように構成されているとき）、あるいはマッチオブジェクトが許可されているメモリ割り当てを消耗しきった場合（Boost.Regex が非再帰モードを使うように構成されているとき）に std::runtime_error。

戻り値: *this。

```
regex_token_iterator& operator++(int);
```

効果: 戻り値用に*this のコピーを構築した後、++(*this)を呼び出す。

戻り値: 結果。

```
template <class charT, class traits>
regex_token_iterator<const charT*, charT, traits>
   make_regex_token_iterator(
         const charT* p,
         const basic_regex<charT, traits>& e,
         int submatch = 0,
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, class ST, class SA>
regex_token_iterator<typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
   make_regex_token_iterator(
         const std::basic_string<charT, ST, SA>& p,
         const basic_regex<charT, traits>& e,
         int submatch = 0,
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, std::size_t N>
```

```
regex_token_iterator<const charT*, charT, traits>
   make_regex_token_iterator(
         const charT* p,
         const basic_regex<charT, traits>& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, class ST, class SA, std::size_t N>
regex_token_iterator<typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
   make_regex_token_iterator(
         const std::basic_string<charT, ST, SA>& p,
         const basic_regex<charT, traits>& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits>
regex_token_iterator<const charT*, charT, traits>
   make_regex_token_iterator(
         const charT* p,
         const basic_regex<charT, traits>& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class traits, class ST, class SA>
regex_token_iterator<
      typename std::basic_string<charT, ST, SA>::const_iterator, charT, traits>
   make_regex_token_iterator(
         const std::basic_string<charT, ST, SA>& p,
         const basic_regex<charT, traits>& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m = regex_constants::match_default);
```

効果: 文字列 *p* 中から正規表現 *e* と match_flag_type *m* を用いて見つかる各マッチに対して、*submatch* 内の値に対応する 1 つの sub_match を列挙する regex_token_iterator を返す。

使用例

次の例は文字列を受け取り、トークン列に分解する。

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

using namespace std;

int main(int argc)
{
   string s;
   do{
      if(argc == 1)
      {
         cout << "分解するテキストを入力してください（\"quit\"で終了）：";
         getline(cin, s);
         if(s == "quit") break;
      }
      else
         s = "This is a string of tokens";
```

```
      boost::regex re("\\s+");
      boost::sregex_token_iterator i(s.begin(), s.end(), re, -1);
      boost::sregex_token_iterator j;

      unsigned count = 0;
      while(i != j)
      {
         cout << *i++ << endl;
         count++;
      }
      cout << "テキスト内に " << count << " 個のトークンが見つかりました。" << endl;

   }while(argc == 1);
   return 0;
}
```

次の例はHTMLファイルを受け取り、リンクしているファイルのリストを出力する。

```
#include <fstream>
#include <iostream>
#include <iterator>
#include <boost/regex.hpp>

boost::regex e("<\\s*A\\s+[^>]*href\\s*=\\s*\"([^\"]*)\"",
               boost::regex::normal | boost::regbase::icase);

void load_file(std::string& s, std::istream& is)
{
   s.erase();
   //
   // ファイルサイズに合わせて文字列バッファを拡張する。
   // 場合によっては正しく動作しない…
   s.reserve(is.rdbuf()->in_avail());
   char c;
   while(is.get(c))
   {
      // （上の）in_avail が 0 を返した場合は
      // 対数拡大法を使う：
      if(s.capacity() == s.size())
         s.reserve(s.capacity() * 3);
      s.append(1, c);
   }
}

int main(int argc, char** argv)
{
   std::string s;
   int i;
   for(i = 1; i < argc; ++i)
   {
      std::cout << "次のファイルで URL を検索中 " << argv[i] << " : " << std::endl;
      s.erase();
      std::ifstream is(argv[i]);
      load_file(s, is);
      boost::sregex_token_iterator i(s.begin(), s.end(), e, 1);
      boost::sregex_token_iterator j;
      while(i != j)
```

```
      {
         std::cout << *i++ << std::endl;
      }
   }
   //
   // 別の方法:
   // 配列直値版コンストラクタのテスト。マッチ全体を
   // $1...同様に分割する
   //
   for(i = 1; i < argc; ++i)
   {
      std::cout << "次のファイルでURLを検索中 " << argv[i] << " : " << std::endl;
      s.erase();
      std::ifstream is(argv[i]);
      load_file(s, is);
      const int subs[] = {1, 0,};
      boost::sregex_token_iterator i(s.begin(), s.end(), e, subs);
      boost::sregex_token_iterator j;
      while(i != j)
      {
         std::cout << *i++ << std::endl;
      }
   }

   return 0;
}
```

bad_expression

概要

```
#include <boost/pattern_except.hpp>
```

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際に発生したエラーを報告するのに投げられる例外オブジェクトの型を定義する。

```
namespace boost{

class regex_error : public std::runtime_error
{
public:
   explicit regex_error(const std::string& s, regex_constants::error_type err, std::ptrdiff_t pos);
   explicit regex_error(boost::regex_constants::error_type err);
   boost::regex_constants::error_type code()const;
   std::ptrdiff_t position()const;
};

typedef regex_error bad_pattern; // 後方互換のため
typedef regex_error bad_expression; // 後方互換のため

} // namespace boost
```

説明

```
regex_error(const std::string& s, regex_constants::error_type err, std::ptrdiff_t pos);
regex_error(boost::regex_constants::error_type err);
```

効果: regex_error クラスのオブジェクトを構築する。

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

効果: 発生した解析エラーを表すエラーコードを返す。

```
std::ptrdiff_t position()const;
```

効果: 解析が停止した正規表現内の位置を返す。

補足: regex_error の基本クラスに std::runtime_error を選択したことについては議論の余地がある。ライブラリの使い方という点では、例外は論理エラー（プログラマが正規表現を与える）、実行時エラー（ユーザが正規表現を与える）のいずれでもよいと考えられる。このライブラリは以前はエラーに bad_pattern と bad_expression を使っていたが、Technical Report on C++ Library Extension と同期をとるために regex_error クラスに一本化した。

syntax_option_type

syntax_option_type の概要

syntax_option_type 型は実装固有のビットマスク型で、正規表現文字列の解釈方法を制御する。利便性のために、ここに挙げる定数はすべて basic_regex テンプレートクラスのスコープにも複製していることに注意していただきたい。

```
namespace std{ namespace regex_constants{

typedef implementation-specific-bitmask-type syntax_option_type;

// 以下のフラグは標準化されている：
static const syntax_option_type normal;
static const syntax_option_type ECMAScript = normal;
static const syntax_option_type JavaScript = normal;
static const syntax_option_type JScript = normal;
static const syntax_option_type perl = normal;
static const syntax_option_type basic;
static const syntax_option_type sed = basic;
static const syntax_option_type extended;
static const syntax_option_type awk;
static const syntax_option_type grep;
static const syntax_option_type egrep;
static const syntax_option_type icase;
static const syntax_option_type nosubs;
static const syntax_option_type optimize;
static const syntax_option_type collate;
```

```
//
// 残りのオプションはBoost.Regex固有のものである：
//

// PerlおよびPOSIX正規表現共通のオプション：
static const syntax_option_type newline_alt;
static const syntax_option_type no_except;
static const syntax_option_type save_subexpression_location;

// Perl固有のオプション：
static const syntax_option_type no_mod_m;
static const syntax_option_type no_mod_s;
static const syntax_option_type mod_s;
static const syntax_option_type mod_x;
static const syntax_option_type no_empty_expressions;

// POSIX拡張固有のオプション：
static const syntax_option_type no_escape_in_lists;
static const syntax_option_type no_bk_refs;

// POSIX基本のオプション：
static const syntax_option_type no_escape_in_lists;
static const syntax_option_type no_char_classes;
static const syntax_option_type no_intervals;
static const syntax_option_type bk_plus_qm;
static const syntax_option_type bk_vbar;

} // namespace regex_constants
} // namespace std
```

syntax_option_type の概観

syntax_option_type 型は実装固有のビットマスク型である（C++標準 17.3.2.1.2 を見よ）。各要素の効果は以下の表に示すとおりである。syntax_option_type 型の値は normal、basic、extended、awk、grep、egrep、sed、literal、perl のいずれか 1 つの要素を必ず含んでいなければならない。

利便性のために、ここに挙げる定数はすべて basic_regex テンプレートクラスのスコープにも複製していることに注意していただきたい。よって、次のコードは、

```
boost::regex_constants::constant_name
```

次のように書くことができる。

```
boost::regex::constant_name
```

あるいは次のようにも書ける。

```
boost::wregex::constant_name
```

以上はいずれも同じ意味である。

Perl 正規表現のオプション

Perl の正規表現では、以下のいずれか 1 つを必ず設定しなければならない。

要素	標準か	設定した場合の効果
ECMAScript	○	正規表現エンジンが解釈する文法が通常のセマンティクスに従うことを指定する。ECMA-262, ECMAScript Language Specification, Chapter 15 part 10, RegExp (Regular Expression) Objects (FWD.1) に与えられているものと同じである。 これは Perl の正規表現構文と機能的には等価である。 このモードでは、Boost.Regex は Perl 互換の (?...) 拡張もサポートする。
perl	×	上に同じ。
normal	×	上に同じ。
JavaScript	×	上に同じ。
JScript	×	上に同じ。

Perl スタイルの正規表現を使用する場合は、以下のオプションを組み合わせることができる。

要素	標準か	設定した場合の効果
icase	○	文字コンテナシーケンスに対する正規表現マッチにおいて、大文字小文字を区別しないことを指定する。
nosubs	○	文字コンテナシーケンスに対して正規表現マッチしたときに、与えられた match_results 構造体に部分式マッチを格納しないように指定する。
optimize	○	正規表現エンジンに対し、正規表現オブジェクトの構築速度よりも正規表現マッチの速度についてより多くの注意を払うように指定する。設定しない場合でもプログラムの出力に検出可能な効果はない。Boost.Regex では現時点では何も起こらない。
collate	○	[a-b] 形式の文字範囲がロカールを考慮するように指定する。
newline_alt	×	\n 文字が選択演算子 | と同じ効果を持つように指定する。これにより、改行で区切られたリストが選択のリストとして動作する。
no_except	×	不正な式が見つかった場合に basic_regex が例外を投げるのを禁止する。
no_mod_m	×	通常 Boost.Regex は Perl の m 修飾子が設定された状態と同じ動作をし、表明 ^ および $ はそれぞれ改行の直前および直後にマッチする。このフラグを設定するのは式の前に (?-m) を追加するのと同じである。
no_mod_s	×	通常 Boost.Regex において . が改行文字にマッチするかはマッチフラグ match_dot_not_newline により決まる。このフラグを設定するのは式の前に (?-s) を追加するのと同じであり、. はマッチフラグに match_dot_not_newline が設定されているかに関わらず改行文字にマッチしない。
mod_s	×	通常 Boost.Regex において . が改行文字にマッチするかはマッチフラグ match_dot_not_newline により決まる。このフラグを設定するのは式の前に (?

要素	標準か	設定した場合の効果
		s)を追加するのと同じであり、.はマッチフラグにmatch_dot_not_newlineが設定されているかに関わらず改行文字にマッチする。
mod_x	×	Perl の x 修飾子を有効にする。正規表現中のエスケープされていない空白は無視される。
no_empty_expressions	×	空の部分式および選択を禁止する。
save_subexpression_location	×	**元の正規表現文字列**における個々の部分式の位置に、basic_regexのsubexpression()メンバ関数でアクセス可能になる。

POSIX 拡張正規表現のオプション

POSIX 拡張正規表現では、以下のいずれか 1 つを必ず設定しなければならない。

要素	標準か	設定した場合の効果
extended	○	正規表現エンジンが IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Base Definitions and Headers, Section 9, Regular Expressions (FWD.1)の POSIX 拡張正規表現で使用されているものと同じ文法に従うことを指定する。 詳細は POSIX 拡張正規表現ガイドを参照せよ。 Perl スタイルのエスケープシーケンスもいくつかサポートする（POSIX 標準の定義では「特殊な」文字のみがエスケープ可能であり、他のエスケープシーケンスを使用したときの結果は未定義である）。
egrep	○	正規表現エンジンが IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Shells and Utilities, Section 4, Utilities, grep (FWD.1)の POSIX ユーティリティに-E オプションを与えた場合と同じ文法に従うことを指定する。 つまり POSIX 拡張構文と同じであるが、改行文字が|と同じく選択文字として動作する。
awk	○	正規表現エンジンが IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Shells and Utilities, Section 4, awk (FWD.1)の POSIX ユーティリティ awk の文法に従うことを指定する。 つまり POSIX 拡張構文と同じであるが、文字クラス中のエスケープシーケンスが許容される。 さらに Perl スタイルのエスケープシーケンスもいくつかサポートする（実際には awk の構文は\a、\b、\t、\v、\f、\nおよび\rのみを要求しており、他のすべての Perl スタイルのエスケープシーケンスを使用したときの動作は未定義であるが、Boost.Regex では実際には後者も解釈する）。

POSIX 拡張正規表現を使用する場合は、以下のオプションを組み合わせることができる。

要素	標準か	設定した場合の効果
icase	○	文字コンテナシーケンスに対する正規表現マッチにおいて、大文字小文字を区別しないことを指定する。
nosubs	○	文字コンテナシーケンスに対して正規表現マッチしたときに、与えられたmatch_results構造体に部分式マッチを格納しないように指定する。
optimize	○	正規表現エンジンに対し、正規表現オブジェクトの構築速度よりも正規表現マッチの速度についてより多くの注意を払うように指定する。設定しない場合でもプログラムの出力に検出可

要素	標準か	設定した場合の効果
		能な効果はない。Boost.Regex では現時点では何も起こらない。
collate	○	[a-b]形式の文字範囲がロカールを考慮するように指定する。このビットは POSIX 拡張正規表現では既定でオンであるが、オフにして範囲をコードポイントのみで比較するようにすることが可能である。
newline_alt	×	\n 文字が選択演算子 | と同じ効果を持つように指定する。これにより、改行で区切られたリストが選択のリストとして動作する。
no_escape_in_lists	×	設定するとエスケープ文字はリスト内で通常の文字として扱われる。よって[\b]は "\" か "b" にマッチする。このビットは POSIX 拡張正規表現では既定でオンであるが、オフにしてリスト内でエスケープが行われるようにすることが可能である。
no_bk_refs	×	設定すると後方参照が無効になる。このビットは POSIX 拡張正規表現では既定でオンであるが、オフにして後方参照を有効にすることが可能である。
no_except	×	不正な式が見つかった場合に basic_regex が例外を投げるのを禁止する。
save_subexpression_location	×	元の正規表現文字列における個々の部分式の位置に、basic_regex の subexpression() メンバ関数でアクセス可能になる。

POSIX 基本正規表現のオプション

POSIX 基本正規表現では、以下のいずれか 1 つを必ず設定しなければならない。

要素	標準か	設定した場合の効果
basic	○	正規表現エンジンが IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Base Definitions and Headers, Section 9, Regular Expressions (FWD.1)の POSIX 基本正規表現で使用されているものと同じ文法に従うことを指定する。
sed	×	上に同じ。
grep	○	正規表現エンジンが IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Shells and Utilities, Section 4, Utilities, grep (FWD.1)の POSIX grep ユーティリティで使用されているものと同じ文法に従うことを指定する。 つまり POSIX 基本構文と同じであるが、改行文字が選択文字として動作する。式は改行区切りの選択リストとして扱われる。
emacs	×	使用する文法が emacs プログラムで使われている POSIX 基本構文のスーパーセットであることを指定する。

POSIX 基本正規表現を使用する場合は、以下のオプションを組み合わせることができる。

要素	標準か	設定した場合の効果
icase	○	文字コンテナシーケンスに対する正規表現マッチにおいて、大文字小文字を区別しないことを指定する。
nosubs	○	文字コンテナシーケンスに対して正規表現マッチしたときに、与えられた match_results 構造体に部分式マッチを格納しないように指定する。
optimize	○	正規表現エンジンに対し、正規表現オブジェクトの構築速度よりも正規表現マッチの速度に

要素	標準か	設定した場合の効果
		ついてより多くの注意を払うように指定する。設定しない場合でもプログラムの出力に検出可能な効果はない。Boost.Regex では現時点では何も起こらない。
collate	○	[a-b]形式の文字範囲がロカールを考慮するように指定する。このビットは POSIX 基本正規表現では既定でオンであるが、オフにして範囲をコードポイントのみで比較するようにすることが可能である。
newline_alt	○	\n 文字が選択演算子 | と同じ効果を持つように指定する。これにより、改行で区切られたリストが選択のリストとしてはたらく。grep オプションの場合はこのビットは常にオンである。
no_char_classes	×	設定すると[[:alnum:]]のような文字クラスは認められないようになる。
no_escape_in_lists	×	設定するとエスケープ文字はリスト内で通常の文字として扱われる。よって [\b] は "\" か "b" にマッチする。このビットは POSIX 基本正規表現では既定でオンであるが、オフにしてリスト内でエスケープが行われるようにすることが可能である。
no_intervals	×	設定すると{2,3}のような境界付き繰り返しは認められないようになる。
bk_plus_qm	×	設定すると\?が 0 か 1 回の繰り返し演算子、\+が 1 回以上の繰り返し演算子として動作する。
bk_vbar	×	設定すると|が選択演算子として動作する。
no_except	×	不正な式が見つかった場合に basic_regex が例外を投げるのを禁止する。
save_subexpression_location	×	元の正規表現文字列における個々の部分式の位置に、basic_regex の subexpression() メンバ関数でアクセス可能になる。

直値文字列のオプション

直値文字列では、以下のいずれか 1 つを必ず設定しなければならない。

要素	標準か	設定した場合の効果
literal	○	文字列を直値として扱う（特殊文字が存在しない）。

literal フラグを使用する場合は、以下のオプションを組み合わせることができる。

要素	標準か	設定した場合の効果
icase	○	文字コンテナシーケンスに対する正規表現マッチにおいて、大文字小文字を区別しないことを指定する。
optimize	○	正規表現エンジンに対し、正規表現オブジェクトの構築速度よりも正規表現マッチの速度についてより多くの注意を払うように指定する。設定しない場合でもプログラムの出力に検出可能な効果はない。Boost.Regex では現時点では何も起こらない。

match_flag_type

match_flag_type 型は実装固有のビットマスク型（C++標準 17.3.2.1.2）で、正規表現の文字シーケンスに対するマッチ方法を制御する。書式化フラグの動作は書式化構文ガイドに詳細を記述する。

```
namespace boost{ namespace regex_constants{
```

```
typedef implemenation-specific-bitmask-type match_flag_type;

static const match_flag_type match_default = 0;
static const match_flag_type match_not_bob;
static const match_flag_type match_not_eob;
static const match_flag_type match_not_bol;
static const match_flag_type match_not_eol;
static const match_flag_type match_not_bow;
static const match_flag_type match_not_eow;
static const match_flag_type match_any;
static const match_flag_type match_not_null;
static const match_flag_type match_continuous;
static const match_flag_type match_partial;
static const match_flag_type match_single_line;
static const match_flag_type match_prev_avail;
static const match_flag_type match_not_dot_newline;
static const match_flag_type match_not_dot_null;
static const match_flag_type match_posix;
static const match_flag_type match_perl;
static const match_flag_type match_nosubs;
static const match_flag_type match_extra;
static const match_flag_type format_default = 0;
static const match_flag_type format_sed;
static const match_flag_type format_perl;
static const match_flag_type format_literal;
static const match_flag_type format_no_copy;
static const match_flag_type format_first_only;
static const match_flag_type format_all;

} // namespace regex_constants
} // namespace boost
```

説明

`match_flag_type` 型は実装固有のビットマスク型（C++標準 17.3.2.1.2）である。文字シーケンス [first, last) に対して正規表現マッチを行うとき、各要素を設定した場合の効果を以下の表に示す。

要素	設定した場合の効果
`match_default`	正規表現マッチを ECMA-262, ECMAScript Language Specification, Chapter 15 part 10, RegExp (Regular Expression) Objects (FWD.1)で使用されている通常の規則にそのまま従うことを指定する。
`match_not_bob`	正規表現`\A`および`\``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_eob`	正規表現`\'`、`\z`および`\Z`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_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`\<`および`\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`\>`および`\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に、それらのいずれでも結果として適合することを指定する。結果が最左マッチとなることには変わりないが、当該位置における最良マッチは保証されない。何がマッ

要素	設定した場合の効果
	チするかよりも速度を優先する場合（マッチがあるかないかのみを調べる場合）にこのフラグを使用するとよい。
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、先頭から始まる部分シーケンスにのみマッチすることを指定する。
match_partial	マッチが見つからない場合に `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）。テキストが不完全であるか非常に長い場合に、このフラグを使用するとよい。詳細は部分マッチの項を見よ。
match_extra	有効な捕捉情報をすべて格納するように正規表現エンジンに指示する。捕捉グループが繰り返しになっている場合、`match_results::captures()` および `sub_match::captures()` を用いて各繰り返しに対する情報にアクセスできる。
match_single_line	Perl の m 修飾子の反転と同様で、`^`が組み込みの改行文字の直後に、`$`が組み込みの改行文字の直前にマッチしないことを指定する（よって、この 2 つのアンカーはそれぞれマッチ対象テキストの先頭、終端にのみマッチする）。
match_prev_avail	`--first` が合法なイテレータ位置であることを指定する。このフラグを設定した場合、正規表現アルゴリズム（RE.7）およびイテレータ（RE.8）はフラグ `match_not_bol` と `match_not_bow` を無視する。[7]
match_not_dot_newline	正規表現`.`が改行文字にマッチしないことを指定する。Perl の s 修飾子の反転と同じである。
match_not_dot_null	正規表現`.`が null 文字 `"\0"` にマッチしないことを指定する。
match_posix	コンパイル済み正規表現の種類に関わらず、POSIX の最左規則にしたがって式のマッチを行うことを指定する。貪欲でない繰り返しなどの Perl 固有の多くの機能を使用する場合、これらの規則は正しく動作しないことに注意していただきたい。
match_perl	コンパイル済み正規表現の種類に関わらず、Perl のマッチ規則にしたがって式のマッチを行うことを指定する。
match_nosubs	実際に捕捉グループが与えられていても、マーク済み部分式が存在しないとして正規表現を扱う。`match_results` クラスにはマッチ全体に関する情報のみ含まれ、部分式については記録されない。
format_default	正規表現マッチを新文字列で置換するとき、ECMA-262, ECMAScript Language Specification, Chapter 15 part 5.4.11 String.prototype.replace. (FWD.1)の ECMAScript replace 関数で使用されている規則を用いて新文字列を構築する。 機能的には Perl の書式化文字列の規則と等価である。 検索・置換操作時に指定すると、正規表現は互いに重複しない位置でマッチし、置換する。正規表現にマッチしなかったテキスト部分はそのまま出力文字列にコピーする。
format_sed	正規表現マッチを新文字列で置換するとき、IEEE Std 1003.1-2001, Portable Operating SystemInterface (POSIX), Shells and Utilities の Unix sed ユーティリティで使用されている規則を用いて新文字列を構築する。sed の書式化文字列リファレンスも見よ。

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

要素	設定した場合の効果
format_perl	正規表現マッチを新文字列で置換するとき、Perl 5と同じ規則を用いて新文字列を生成する。
format_literal	正規表現マッチを新文字列で置換するとき、置換テキストの直値コピーを新文字列とする。
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。詳細は書式化文字列のガイドを見よ。
format_no_copy	検索・置換操作時に指定すると、検索対象の文字コンテナシーケンスの正規表現にマッチしない部分を出力文字列にコピーしない。
format_first_only	検索・置換操作時に指定すると、最初の正規表現マッチのみを置換する。

error_type

概要

型 error_type は、正規表現解析時にライブラリが発生させる可能性のある様々な種類のエラーを表す。

```
namespace boost{ namespace regex_constants{

typedef implementation-specific-type error_type;

static const error_type error_collate;
static const error_type error_ctype;
static const error_type error_escape;
static const error_type error_backref;
static const error_type error_brack;
static const error_type error_paren;
static const error_type error_brace;
static const error_type error_badbrace;
static const error_type error_range;
static const error_type error_space;
static const error_type error_badrepeat;
static const error_type error_complexity;
static const error_type error_stack;
static const error_type error_bad_pattern;

} // namespace regex_constants
} // namespace boost
```

説明

型 error_type は以下のいずれかの値をとる実装固有の列挙型である。

定数	意味
error_collate	[[.name.]]ブロックで指定した照合要素が不正。
error_ctype	[[:name:]]ブロックで指定した文字クラス名が不正。
error_escape	不正なエスケープか本体のないエスケープが見つかった。
error_backref	存在しないマーク済み部分式への後方参照が見つかった。
error_brack	不正な文字集合[…]が見つかった。

定数	意味
error_paren	(と)が正しく対応していない。
error_brace	{と}が正しく対応していない。
error_badbrace	{…}ブロックの内容が不正。
error_range	文字範囲が不正（例 [d-a]）。
error_space	メモリ不足。
error_badrepeat	繰り返し不能なものを繰り返そうとした（例 a*+）。
error_complexity	式が複雑で処理できなかった。
error_stack	プログラムのスタック空間不足。
error_bad_pattern	その他のエラー。

regex_traits

```
namespace boost{

template <class charT, class implementationT = sensible_default_choice>
struct regex_traits : public implementationT
{
  regex_traits() : implementationT() {}
};

template <class charT>
struct c_regex_traits;

template <class charT>
class cpp_regex_traits;

template <class charT>
class w32_regex_traits;

} // namespace boost
```

説明

regex_traits クラスは以下のいずれかである、実装クラスの薄いラッパである。

- c_regex_traits：このクラスは非推奨である。C ロカールをラップし、Win32 以外のプラットフォームで C++ロカールが利用不能な場合に使用される。
- cpp_regex_traits：非 Win32 プラットフォームにおける既定の特性クラスである。正規表現クラスのロカールを変更するのに std::locale インスタンスが使用可能である。
- w32_regex_traits：Win32 プラットフォームにおける既定の特性クラスである。正規表現クラスのロカールを変更するのに LCID が使用可能である。

既定の動作は boost/regex/user.hpp にある以下の設定マクロのいずれかを定義することで変更可能である。

- BOOST_REGEX_USE_C_LOCALE：c_regex_traits が既定となる。
- BOOST_REGEX_USE_CPP_LOCALE：cpp_regex_traits が既定となる。

これらの特性クラスは特性クラスの要件を満たす。

非標準文字列型に対するインターフェイス

Boost.Regex のアルゴリズムおよびイテレータはすべてイテレータベースである。また、標準ライブラリの文字列型を内部でイテレータの組に変換する、便利なアルゴリズムの多重定義を提供する。標準以外の文字列型に対して検索を行うのであれば、その文字列をイテレータの組に変換すればよい。イテレータベースでないとされているものも含めて、私はこの方法で処理不能な文字列型を見たことがない。もちろん、内部バッファと長さへのアクセスを提供する文字列型は、すべて（イテレータの組として使用可能である）ポインタの組に変換可能である。

標準以外の文字列型の中でも、広く使われているため既にラッパが用意されているものがある。現在は ICU と MFC の文字列クラス型にラッパがある。

Unicode と ICU 文字列型

ICU とともに Boost.Regex を使用する

ヘッダ

```
<boost/regex/icu.hpp>
```

に、Unicode 環境で正規表現を使用するのに必要なデータ型とアルゴリズムが含まれている。

このヘッダを使用する場合は ICU ライブラリが必要である。また ICU サポートを有効にして Boost.Regex をビルドしていなければならない。

このヘッダにより以下のことが可能となる。

- Unicode 文字列を UTF-32 コードポイントシーケンスとして扱う正規表現の作成。
- 文字分類を含む Unicode データプロパティをサポートする正規表現の作成。
- UTF-8、UTF-16、UTF-32 のいずれかで符号化された Unicode 文字列の透過的な検索。

Unicode 正規表現型

ヘッダ <boost/regex/icu.hpp> は UTF-32 文字を処理する正規表現特性クラスを提供する。

```
class icu_regex_traits;
```

そしてこの特性クラスを用いた正規表現型がある。

```
typedef basic_regex<UChar32,icu_regex_traits> u32regex;
```

型 u32regex はあらゆる Unicode 正規表現を使用するための正規表現型である。内部的には UTF-32 コードポイントを使用しているが、UTF-32 符号化文字列だけでなく、UTF-8 および UTF-16 符号化文字列による作成・検索も可能である。

u32regex のコンストラクタおよび assign メンバ関数は UTF-32 符号化文字列を要求するが、UTF-8、UTF-16 および UTF-32 符号化文字列から正規表現を作成する make_u32regex アルゴリズムの多重定義群がある。

```
template <class InputIterator>
u32regex make_u32regex(InputIterator i,
                       InputIterator j,
                       boost::regex_constants::syntax_option_type opt);
```

効果: イテレータシーケンス [i,j) から正規表現オブジェクトを作成する。シーケンスの文字符号化形式は sizeof(*i) により決定し、1 であれば UTF-8、2 であれば UTF-16、4 であれば UTF-32 となる。

```
u32regex make_u32regex(const char* p,
                       boost::regex_constants::syntax_option_type opt
                           = boost::regex_constants::perl);
```

効果: null 終端 UTF-8 文字シーケンス *p* から正規表現オブジェクトを作成する。

```
u32regex make_u32regex(const unsigned char* p,
                       boost::regex_constants::syntax_option_type opt
                           = boost::regex_constants::perl);
```

効果: null 終端 UTF-8 文字シーケンス *p* から正規表現オブジェクトを作成する。

```
u32regex make_u32regex(const wchar_t* p,
                       boost::regex_constants::syntax_option_type opt
                           = boost::regex_constants::perl);
```

効果: null 終端文字シーケンス *p* から正規表現オブジェクトを作成する。シーケンスの文字符号化形式 sizeof(wchar_t) により決定し、1 であれば UTF-8、2 であれば UTF-16、4 であれば UTF-32 となる。

```
u32regex make_u32regex(const UChar* p,
                       boost::regex_constants::syntax_option_type opt
                           = boost::regex_constants::perl);
```

効果: null 終端 UTF-16 文字シーケンス *p* から正規表現オブジェクトを作成する。

```
template<class C, class T, class A>
u32regex make_u32regex(const std::basic_string<C, T, A>& s,
                       boost::regex_constants::syntax_option_type opt
                           = boost::regex_constants::perl);
```

効果: 文字列 s から正規表現オブジェクトを作成する。シーケンスの文字符号化形式 `sizeof(C)` により決定し、1 であれば UTF-8、2 であれば UTF-16、4 であれば UTF-32 となる。

```
u32regex make_u32regex(const UnicodeString& s,
                       boost::regex_constants::syntax_option_type opt
                           = boost::regex_constants::perl);
```

効果: UTF-16 符号化文字列 *s* から正規表現オブジェクトを作成する。

Unicode 正規表現アルゴリズム

正規表現アルゴリズム regex_match、regex_search および regex_replace はすべて、処理する文字シーケンスの文字エンコーディングが正規表現オブジェクトで使われているものと同じであると想定している。この動作は Unicode 正規表現では望ましいものではない。[8]データを UTF-32 の「チャンク」で処理したくても、実際のデータは UTF-8 か UTF-16 で符号化されている場合が多い。そのためヘッダ <boost/regex/icu.hpp>はこれらのアルゴリズムの薄いラッパ群 `u32regex_match`、`u32regex_search` および `u32regex_replace` を提供している。これらのラッパは内部でイテレータアダプタを使って、実際は「本体の」アルゴリズムに渡すことのできる UTF-32 シーケンスであるデータを見かけ上 UTF-8、UTF-16 としている。

u32regex_match

各 regex_match アルゴリズムが<boost/regex.hpp>で定義されているのに対し、<boost/regex/icu.hpp>は同じ引数をとる多重定義アルゴリズム `u32regex_match` を定義する。入力として ICU の `UnicodeString` とともに UTF-8、UTF-16、UTF-32 符号化データを受け取る。

例:パスワードのマッチを UTF-16 `UnicodeString` で行う。

```
//
// password が正規表現 requirements で
// 定義したパスワードの要件を満たしているか調べる。
//
bool is_valid_password(const UnicodeString& password, const UnicodeString& requirements)
{
   return boost::u32regex_match(password, boost::make_u32regex(requirements));
}
```

例:UTF-8 で符号化されたファイル名のマッチを行う。

```
//
// UTF-8 で符号化された std::string のパスからファイル名部分を抜き出し、
// 結果を別の std::string として返す：
//
std::string get_filename(const std::string& path)
{
   boost::u32regex r = boost::make_u32regex("(?:\\A|.*\\\\)([^\\\\]+)");
   boost::smatch what;
   if(boost::u32regex_match(path, what, r))
```

8 訳注 Unicode に限った話ではありません。日本語では従来から複数の符号化方式を使用しています。

```
   {
      // $1 を std::string として抽出する：
      return what.str(1);
   }
   else
   {
      throw std::runtime_error("パス名が不正");
   }
}
```

u32regex_search

各 regex_search アルゴリズムが<boost/regex.hpp>で定義されているのに対し、<boost/regex/icu.hpp>は同じ引数をとる多重定義アルゴリズム u32regex_search を定義する。入力として ICU の UnicodeString とともに UTF-8、UTF-16、UTF-32 符号化データを受け取る。

例：特定の言語区画から文字シーケンスを検索する。

```
UnicodeString extract_greek(const UnicodeString& text)
{
   // UTF-16 で符号化されたテキストからギリシャ語の区画を検索する。
   // この正規表現は完全ではないが、今のところは最善の方法である。特定の
   // 用字系を検索するのは、実際は非常に難しい。
   //
   // 検索するのはギリシャ文字で始まり
   // 非アルファベット（[^[:L*:]]）かギリシャ文字ブロック
   // （[\\x{370}-\\x{3FF}]）の文字が続く文字シーケンスである。
   //
   boost::u32regex r = boost::make_u32regex(
         L"[\\x{370}-\\x{3FF}](?:[^[:L*:]]|[\\x{370}-\\x{3FF}])*");
   boost::u16match what;
   if(boost::u32regex_search(text, what, r))
   {
      // $0 を UnicodeString として抽出する：
      return UnicodeString(what[0].first, what.length(0));
   }
   else
   {
      throw std::runtime_error("ギリシャ語の部分は見つかりませんでした！");
   }
}
```

u32regex_replace

各 regex_replace アルゴリズムが<boost/regex.hpp>で定義されているのに対し、<boost/regex/icu.hpp>は同じ引数をとる多重定義アルゴリズム u32regex_replace を定義する。入力として ICU の UnicodeString とともに UTF-8、UTF-16、UTF-32 符号化データを受け取る。アルゴリズムに渡す入力シーケンスと書式化文字列の符号化形式は異なっていてもよい（一方が UTF-8 で他方が UTF-16 など）が、結果の文字列や出力イテレータは検索対象のテキストと同じ文字符号化形式でなければならない。

例：クレジットカード番号を書式化しなおす。

```
//
// クレジットカード番号を（数字を含んだ）文字列として受け取り、
// 4 桁ずつ "-" で区切られた可読性の高い形式に
// 再書式化する。
// UTF-32 の正規表現、UTF-16 の文字列、
// UTF-8 の書式指定子を混在させているが
// すべて正しく動作することに注意していただきたい：
//
const boost::u32regex e = boost::make_u32regex(
      "\\A(\\d{3,4})[- ]?(\\d{4})[- ]?(\\d{4})[- ]?(\\d{4})\\z");
const char* human_format = "$1-$2-$3-$4";

UnicodeString human_readable_card_number(const UnicodeString& s)
{
   return boost::u32regex_replace(s, e, human_format);
}
```

Unicode 正規表現イテレータ

u32regex_iterator

型 u32regex_iterator はあらゆる側面で regex_iterator と同じであるが、正規表現型が常に u32regex であることからテンプレート引数を 1 つ（イテレータ型）だけとる点が異なる。内部で u32regex_search を呼び出し、UTF-8、UTF-16 および UTF-32 のデータを正しく処理する。

```
template <class BidirectionalIterator>
class u32regex_iterator
{
   // メンバについては regex_iterator を参照
};

typedef u32regex_iterator<const char*>     utf8regex_iterator;
typedef u32regex_iterator<const UChar*>    utf16regex_iterator;
typedef u32regex_iterator<const UChar32*>  utf32regex_iterator;
```

文字列から u32regex_iterator を簡単に構築するために、非メンバのヘルパ関数群 make_u32regex_iterator がある。

```
u32regex_iterator<const char*>
   make_u32regex_iterator(const char* s,
                          const u32regex& e,
                          regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_iterator<const wchar_t*>
   make_u32regex_iterator(const wchar_t* s,
                          const u32regex& e,
                          regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_iterator<const Uchar*>
   make_u32regex_iterator(const UChar* s,
                          const u32regex& e,
                          regex_constants::match_flag_type m = regex_constants::match_default);
```

```
template <class charT, class Traits, class Alloc>
u32regex_iterator<typename std::basic_string<charT, Traits, Alloc>::const_iterator>
   make_u32regex_iterator(const std::basic_string<charT, Traits, Alloc>& s,
                          const u32regex& e,
                          regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_iterator<const Uchar*>
   make_u32regex_iterator(const UnicodeString& s,
                          const u32regex& e,
                          regex_constants::match_flag_type m = regex_constants::match_default);
```

これらの多重定義は、テキスト *s* に対してフラグ *m* を用いて見つかる正規表現 *e* のすべてのマッチを列挙するイテレータを返す。

例：国際通貨記号とその金額（数値）を検索する。

```
void enumerate_currencies(const std::string& text)
{
   // 通貨記号とその金額（数値）を
   // すべて列挙、印字する：
   const char* re =
      "([[:Sc:]][[:Cf:][:Cc:][:Z*:]]*)?"
      "([[:Nd:]]+(?:[[:Po:]][[:Nd:]]+)?)?"
      "(?(1)"
         "|(?(2)"
            "[[:Cf:][:Cc:][:Z*:]]*"
         ")"
         "[[:Sc:]]"
      ")";
   boost::u32regex r = boost::make_u32regex(re);
   boost::u32regex_iterator<std::string::const_iterator>
         i(boost::make_u32regex_iterator(text, r)), j;
   while(i != j)
   {
      std::cout << (*i)[0] << std::endl;
      ++i;
   }
}
```

次のように呼び出すと、

```
enumerate_currencies(" $100.23 or £198.12 ");
```

以下の結果を得る。

```
$100.23
£198.12
```

当然ながら、入力は UTF-8 で符号化したものである。

u32regex_token_iterator

型 `u32regex_token_iterator` はあらゆる側面で `regex_token_iterator` と同じであるが、正規表現型が常に `u32regex` であることからテンプレート引数を 1 つ（イテレータ型）だけとる点が異なる。内部で `u32regex_search` を呼び出し、UTF-8、UTF-16

および UTF-32 のデータを正しく処理する。

```
template <class BidirectionalIterator>
class u32regex_token_iterator
{
   // メンバについては regex_token_iterator を参照
};

typedef u32regex_token_iterator<const char*>     utf8regex_token_iterator;
typedef u32regex_token_iterator<const UChar*>    utf16regex_token_iterator;
typedef u32regex_token_iterator<const UChar32*>  utf32regex_token_iterator;
```

文字列から u32regex_token_iterator を簡単に構築するために、非メンバのヘルパ関数群 make_u32regex_token_iterator がある。

```
u32regex_token_iterator<const char*>
   make_u32regex_token_iterator(
         const char* s,
         const u32regex& e,
         int sub,
         regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_token_iterator<const wchar_t*>
   make_u32regex_token_iterator(
         const wchar_t* s,
         const u32regex& e,
         int sub,
         regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_token_iterator<const Uchar*>
   make_u32regex_token_iterator(
         const UChar* s,
         const u32regex& e,
         int sub,
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class Traits, class Alloc>
u32regex_token_iterator<typename std::basic_string<charT, Traits, Alloc>::const_iterator>
   make_u32regex_token_iterator(
         const std::basic_string<charT, Traits, Alloc>& s,
         const u32regex& e,
         int sub,
         regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_token_iterator<const Uchar*>
   make_u32regex_token_iterator(
         const UnicodeString& s,
         const u32regex& e,
         int sub,
         regex_constants::match_flag_type m = regex_constants::match_default);
```

これらの多重定義は、テキスト *s* に対してフラグ *m* を用いて見つかる正規表現 *e* の部分式 *sub* のすべてのマッチを列挙するイテレータを返す。

```
template <std::size_t N>
```

```
u32regex_token_iterator<const char*>
   make_u32regex_token_iterator(
         const char* p,
         const u32regex& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);

template <std::size_t N>
u32regex_token_iterator<const wchar_t*>
   make_u32regex_token_iterator(
         const wchar_t* p,
         const u32regex& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);

template <std::size_t N>
u32regex_token_iterator<const Uchar*>
   make_u32regex_token_iterator(
         const UChar* p,
         const u32regex& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class Traits, class Alloc, std::size_t N>
u32regex_token_iterator<typename std::basic_string<charT, Traits, Alloc>::const_iterator>
   make_u32regex_token_iterator(
         const std::basic_string<charT, Traits, Alloc>& p,
         const u32regex& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);

template <std::size_t N>
u32regex_token_iterator<const Uchar*>
   make_u32regex_token_iterator(
         const UnicodeString& s,
         const u32regex& e,
         const int (&submatch)[N],
         regex_constants::match_flag_type m = regex_constants::match_default);
```

これらの多重定義は、テキスト *s* に対してフラグ *m* を用いて見つかる正規表現 *e* のすべての部分式マッチを列挙するイテレータを返す。

```
u32regex_token_iterator<const char*>
   make_u32regex_token_iterator(
         const char* p,
         const u32regex& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_token_iterator<const wchar_t*>
   make_u32regex_token_iterator(
         const wchar_t* p,
         const u32regex& e,
         const std::vector<int>& submatch,
         regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_token_iterator<const Uchar*>
   make_u32regex_token_iterator(
```

```
        const UChar* p,
        const u32regex& e,
        const std::vector<int>& submatch,
        regex_constants::match_flag_type m = regex_constants::match_default);

template <class charT, class Traits, class Alloc>
u32regex_token_iterator<typename std::basic_string<charT, Traits, Alloc>::const_iterator>
   make_u32regex_token_iterator(
        const std::basic_string<charT, Traits, Alloc>& p,
        const u32regex& e,
        const std::vector<int>& submatch,
        regex_constants::match_flag_type m = regex_constants::match_default);

u32regex_token_iterator<const Uchar*>
   make_u32regex_token_iterator(
        const UnicodeString& s,
        const u32regex& e,
        const std::vector<int>& submatch,
        regex_constants::match_flag_type m = regex_constants::match_default);
```

これらの多重定義は、テキスト *s* に対してフラグ *m* を用いて見つかる正規表現 *e* の 1 つの部分式マッチを列挙するイテレータを返す。

例：国際通貨記号とその金額（数値）を検索する。

```
void enumerate_currencies2(const std::string& text)
{
   // 通貨記号とその金額（数値）を
   // すべて列挙、印字する：
   const char* re =
      "([[:Sc:]][[:Cf:][:Cc:][:Z*:]]*)?"
      "([[:Nd:]]+(?:[[:Po:]][[:Nd:]]+)?)?"
      "(?(1)"
         "|(?(2)"
            "[[:Cf:][:Cc:][:Z*:]]*"
         ")"
         "[[:Sc:]]"
      ")";
   boost::u32regex r = boost::make_u32regex(re);
   boost::u32regex_token_iterator<std::string::const_iterator>
      i(boost::make_u32regex_token_iterator(text, r, 1)), j;
   while(i != j)
   {
      std::cout << *i << std::endl;
      ++i;
   }
}
```

MFC 文字列とともに Boost.Regex を使用する

はじめに

ヘッダ <boost/regex/mfc.hpp> に、MFC 文字列型に対する Boost.Regex のサポートがある。この機能には、MFC および ATL のすべての文字列型が CSimpleStringT テンプレートクラスに基づいている Visual Studio .NET（Visual C++ 7）以降が必要である

ことに注意していただきたい。

以下の説明では、CSimpleStringT<charT>の箇所は次の MFC/ATL の型に置き換えて読んでかまわない（すべて CSimpleStringT<charT>を継承している）。

```
CString
CStringA
CStringW
CAtlString
CAtlStringA
CAtlStringW
CStringT<charT,traits>
CFixedStringT<charT,N>
CSimpleStringT<charT>
```

MFC 文字列で使用する Boost.Regex の型

TCHAR を使用するのに便利なように、以下の typedef を提供している。

```
typedef basic_regex<TCHAR>                  tregex;
typedef match_results<TCHAR const*>         tmatch;
typedef regex_iterator<TCHAR const*>        tregex_iterator;
typedef regex_token_iterator<TCHAR const*>  tregex_token_iterator;
```

TCHAR ではなく、ナロー文字かワイド文字を明示的に使用する場合は、通常の Boost.Regex 型である regex か wregex を使用するとよい。

MFC 文字列からの正規表現の作成

以下のヘルパ関数は MFC/ATL 文字列型からの正規表現作成を補助する。

```
template <class charT>
basic_regex<charT>
   make_regex(const ATL::CSimpleStringT<charT>& s,
              ::boost::regex_constants::syntax_option_type f = boost::regex_constants::normal);
```

効果:basic_regex<charT>(s.GetString(), s.GetString() + s.GetLength(), f);を返す。

MFC 文字列型に対するアルゴリズムの多重定義

std::basic_string 引数についてのアルゴリズムの各多重定義に対して、MFC/ATL 文字列型についての多重定義がある。これらのアルゴリズムのすべてのシグニチャは実際よりもかなり複雑に見えるが、完全性のためにここではすべて記述する。

regex_match

2 つの多重定義がある。1 番目のものは何がマッチしたかを match_results 構造体で返し、2 番目は何も返さない。

regex_match についての注意がすべてこの関数にも適用されるが、特にこのアルゴリズムは入力テキスト全体の式に対するマッチが成功したかだけを返す。この動作が希望のものでない場合は regex_search を代わりに使用するとよい。

```
template <class charT, class T, class A>
bool regex_match(
   const ATL::CSimpleStringT<charT>& s,
   match_results<const B*, A>& what,
   const basic_regex<charT, T>& e,
   boost::regex_constants::match_flag_type f = boost::regex_constants::match_default);
```

効果:::boost::regex_match(s.GetString(), s.GetString() + s.GetLength(), what, e, f);を返す。

使用例:

```
//
// CStringのパスからファイル名部分を抜き出し、
// 結果をCStringで返す：
//
CString get_filename(const CString& path)
{
   boost::tregex r(__T("(?:\\A|.*\\\\)([^\\\\]+)"));
   boost::tmatch what;
   if(boost::regex_match(path, what, r))
   {
      // $1をCStringとして抽出する：
      return CString(what[1].first, what.length(1));
   }
   else
   {
      throw std::runtime_error("パス名が不正");
   }
}
```

regex_match（第二の多重定義）

```
template <class charT, class T>
bool regex_match(
   const ATL::CSimpleStringT<charT>& s,
   const basic_regex<B, T>& e,
   boost::regex_constants::match_flag_type f = boost::regex_constants::match_default);
```

効果:::boost::regex_match(s.GetString(), s.GetString() + s.GetLength(), e, f);を返す。

使用例:

```
//
// passwordが正規表現requirementsで
// 定義したパスワードの要件を満たしているか調べる。
//
bool is_valid_password(const CString& password, const CString& requirements)
{
   return boost::regex_match(password, boost::make_regex(requirements));
}
```

regex_search

regex_search については多重定義を 2 つ追加する。1 番目のものは何がマッチしたかを返し、2 番目は何も返さない。

```
template <class charT, class A, class T>
bool regex_search(const ATL::CSimpleStringT<charT>& s,
   match_results<const charT*, A>& what,
   const basic_regex<charT, T>& e,
   boost::regex_constants::match_flag_type f = boost::regex_constants::match_default);
```

効果: ::boost::regex_search(s.GetString(), s.GetString() + s.GetLength(), what, e, f);を返す。

使用例: 住所の文字列から郵便番号を抜き出す。

```
CString extract_postcode(const CString& address)
{
   // 投函住所から英国郵便番号を検索し、結果を返す。
   // 正規表現は on www.regxlib.com の Phil A.のものを用いた：
   boost::tregex r(__T("^(([A-Z]{1,2}[0-9]{1,2})|([A-Z]{1,2}[0-9][A-Z]))\\s?([0-9][A-Z]{2})$"));
   boost::tmatch what;
   if(boost::regex_search(address, what, r))
   {
      // $0 を CString として抽出する：
      return CString(what[0].first, what.length());
   }
   else
   {
      throw std::runtime_error("郵便番号は見つかりません");
   }
}
```

regex_search（第二の多重定義）

```
template <class charT, class T>
inline bool regex_search(const ATL::CSimpleStringT<charT>& s,
               const basic_regex<charT, T>& e,
               boost::regex_constants::match_flag_type f = boost::regex_constants::match_default);
```

効果: ::boost::regex_search(s.GetString(), s.GetString() + s.GetLength(), e, f);を返す。

regex_replace

regex_replace については多重定義を 2 つ追加する。1 番目のものは出力イテレータに出力を送り、2 番目は何も出力しない。

```
template <class OutputIterator, class BidirectionalIterator, class traits, class
          charT>
OutputIterator regex_replace(OutputIterator out,
                             BidirectionalIterator first,
                             BidirectionalIterator last,
                             const basic_regex<charT, traits>& e,
                             const ATL::CSimpleStringT<charT>& fmt,
                             match_flag_type flags = match_default);
```

効果:::boost::regex_replace(out, first, last, e, fmt.GetString(), flags);を返す。

```
template <class traits, class charT>
ATL::CSimpleStringT<charT> regex_replace(const ATL::CSimpleStringT<charT>& s,
                           const basic_regex<charT, traits>& e,
                           const ATL::CSimpleStringT<charT>& fmt,
                           match_flag_type flags = match_default);
```

効果:regex_replace、および文字列 *s* と同じメモリマネージャを使って新文字列を作成し、返す。

使用例:

```
//
// クレジットカード番号を（数字を含んだ）文字列で受け取り、
// 4 桁ずつ "-" で区切った可読性の高い形式に
// 再書式化する：
//
const boost::tregex e(__T("\\A(\\d{3,4})[- ]?(\\d{4})[- ]?(\\d{4})[- ]?(\\d{4})\\z"));
const CString human_format = __T("$1-$2-$3-$4");

CString human_readable_card_number(const CString& s)
{
   return boost::regex_replace(s, e, human_format);
}
```

MFC 文字列に対するマッチの反復

MFC/ATL 文字列を regex_iterator および regex_token_iterator に簡単に変換できるように、以下のヘルパ関数を提供する。

regex_iterator 作成ヘルパ

```
template <class charT>
regex_iterator<charT const*>
   make_regex_iterator(
      const ATL::CSimpleStringT<charT>& s,
      const basic_regex<charT>& e,
      ::boost::regex_constants::match_flag_type f = boost::regex_constants::match_default);
```

効果:regex_iterator(s.GetString(), s.GetString() + s.GetLength(), e, f);を返す。

使用例:

```
void enumerate_links(const CString& html)
{
   // HTML テキスト中のリンクをすべて列挙、印字する。
   // 正規表現は www.regxlib.com の Andrew Lee のものを用いた：
   boost::tregex r(
      __T("href=[\"\']((http:\\/\\/|\\.\\/|\\/)?\\w+"
          "(\\.\\w+)*(\\/\\w+(\\.\\w+)?)*"
          "(\\/|\\?\\w*=\\w*(&\\w*=\\w*)*)?)[\"\']"));
```

```
   boost::tregex_iterator i(boost::make_regex_iterator(html, r)), j;
   while(i != j)
   {
      std::cout << (*i)[1] << std::endl;
      ++i;
   }
}
```

regex_token_iterator 作成ヘルパ

```
template <class charT>
regex_token_iterator<charT const*>
   make_regex_token_iterator(
      const ATL::CSimpleStringT<charT>& s,
      const basic_regex<charT>& e,
      int sub = 0,
      ::boost::regex_constants::match_flag_type f = boost::regex_constants::match_default);
```

効果: `regex_token_iterator(s.GetString(), s.GetString() + s.GetLength(), e, sub, f);` を返す。

```
template <class charT>
regex_token_iterator<charT const*>
   make_regex_token_iterator(
      const ATL::CSimpleStringT<charT>& s,
      const basic_regex<charT>& e,
      const std::vector<int>& subs,
      ::boost::regex_constants::match_flag_type f = boost::regex_constants::match_default);
```

効果: `regex_token_iterator(s.GetString(), s.GetString() + s.GetLength(), e, subs, f);` を返す。

```
template <class charT, std::size_t N>
regex_token_iterator<charT const*>
   make_regex_token_iterator(
      const ATL::CSimpleStringT<charT>& s,
      const basic_regex<charT>& e,
      const int (& subs)[N],
      ::boost::regex_constants::match_flag_type f = boost::regex_constants::match_default);
```

効果: `regex_token_iterator(s.GetString(), s.GetString() + s.GetLength(), e, subs, f);` を返す。

使用例:

```
void enumerate_links2(const CString& html)
{
   // HTMLテキスト中のリンクをすべて列挙、印字する。
   // 正規表現はwww.regxlib.comのAndrew Leeのものを用いた：
   boost::tregex r(
         __T("href=[\"\']((http:\\/\\/|\\.\\/|\\/)?\\w+"
             "(\\.\\w+)*(\\/\\w+(\\.\\w+)?)*"
             "(\\/|\\?\\w*=\\w*(&\\w*=\\w*)*)?)[\"\']"));
            boost::tregex_token_iterator i(boost::make_regex_token_iterator(html, r, 1)), j;
   while(i != j)
```

```
   {
      std::cout << *i << std::endl;
      ++i;
   }
}
```

POSIX 互換 C API

注意

本リファレンスは POSIX API 関数の要約である。これらは（C++以外の言語によるアクセスが必要でない限り）新しいコードで使用する API ではなく、他のライブラリとの互換性のために提供されている。これらの関数が使用している名前は実際の関数名に展開されるマクロであるため、他のバージョンとの共存が可能である。

```
#include <boost/cregex.hpp>
```

あるいは

```
#include <boost/regex.h>
```

以下の関数は POSIX 互換の C ライブラリが必要なユーザ向けである。Unicode 版とナロー文字版の両方が利用可能であり、標準 POSIX API 名は UNICODE が定義されているかどうかでいずれかの版に展開されるマクロである。

重要

ここで定義するシンボルは、C++プログラムではすべて名前空間 boost 内にあることに注意していただきたい。#include <boost/regex.h>を使用した場合はシンボルが名前空間 boost 内で定義されるのは変わらないが、大域名前空間でも利用できるようになっている。

関数の定義は以下のとおりである。

```
extern "C" {

struct regex_tA;
struct regex_tW;

int regcompA(regex_tA*, const char*, int);
unsigned int regerrorA(int, const regex_tA*, char*, unsigned int);
int regexecA(const regex_tA*, const char*, unsigned int, regmatch_t*, int);
void regfreeA(regex_tA*);
```

```
int regcompW(regex_tW*, const wchar_t*, int);
unsigned int regerrorW(int, const regex_tW*, wchar_t*, unsigned int);
int regexecW(const regex_tW*, const wchar_t*, unsigned int, regmatch_t*, int);
void regfreeW(regex_tW*);

#ifdef UNICODE
#define regcomp regcompW
#define regerror regerrorW
#define regexec regexecW
#define regfree regfreeW
#define regex_t regex_tW
#else
#define regcomp regcompA
#define regerror regerrorA
#define regexec regexecA
#define regfree regfreeA
#define regex_t regex_tA
#endif
}
```

これらの関数はすべて構造 regex_t に対して処理を行う。この構造体は次の 2 つの公開メンバを持つ。

メンバ	意味
unsigned int re_nsub	regcomp により値が設定され、正規表現中の部分式の総数を表す。
const TCHAR* re_endp	フラグ REG_PEND が設定されている場合、コンパイルする正規表現の終端を指す。

注意

regex_t は実際は#define であり、UNICODE が定義されているかどうかにより regex_tA か regex_tW のいずれかとなる。TCHAR はマクロ UNICODE により char か wchar_t のいずれかとなる。

regcomp

regcomp は regex_t へのポインタ、コンパイルする式へのポインタおよび以下の組み合わせとなるフラグ引数をとる。

フラグ	意味
REG_EXTENDED	現代的な正規表現をコンパイルする。regbase::char_classes | regbase::intervals | regbase::bk_refs と等価である。
REG_BASIC	基本的な（旧式の）正規表現構文をコンパイルする。regbase::char_classes | regbase::intervals | regbase::limited_ops | regbase::bk_braces | regbase::bk_parens | regbase::bk_refs と等価である。
REG_NOSPEC	文字をすべて通常の文字として扱う。正規表現は直値文字列である。
REG_ICASE	大文字小文字を区別しないマッチを行う。
REG_NOSUB	このライブラリでは効果なし。

フラグ	意味
REG_NEWLINE	このフラグを設定した場合、ドットが改行文字にマッチしない。
REG_PEND	このフラグを設定した場合、regex_t 構造体の re_endp 引数はコンパイルする正規表現の終端を指していなければならない。
REG_NOCOLLATE	このフラグを設定した場合、文字範囲においてロカール依存の照合が無効になる。
REG_ESCAPE_IN_LISTS	このフラグを設定した場合、括弧式（文字集合）内でエスケープシーケンスが使用できる。
REG_NEWLINE_ALT	このフラグを設定した場合、改行文字は選択演算子 | と等価である。
REG_PERL	Perl 似の正規表現をコンパイルする。
REG_AWK	awk 似動作のショートカット：REG_EXTENDED | REG_ESCAPE_IN_LISTS
REG_GREP	grep 似動作のショートカット：REG_BASIC | REG_NEWLINE_ALT
REG_EGREP	egrep 似動作のショートカット：REG_EXTENDED | REG_NEWLINE_ALT

regerror

regerror は以下の引数をとり、エラーコードを可読性の高い文字列に変換する。

引数	意味
int code	エラーコード。
const regex_t* e	正規表現（null でもよい）。
char* buf	エラーメッセージを書き込む文字列。
unsigned int buf_size	buf の長さ。

エラーコードが REG_ITOA との論理和になっている場合は、結果はメッセージではなく、例えば “REG_BADPAT” のようなコードの印字可能な名前となる。コードが REG_ATOI の場合は、*e* は null であってはならず e->re_endp は印字可能名の終端を指していなければならない。またこの場合の戻り値はエラーコードの値である。*code* の値がこれら以外の場合は、戻り値はエラーメッセージの文字数であり、戻り値が *buf_size* 以上であればより大きなバッファを用いて regerror を再度呼び出す必要がある。

regexec

regexec は文字列 *buf* 内から式 *e* の最初のマッチを検索する。*len* が 0 以外の場合は、**m* には正規表現にマッチした内容が書き込まれる。m[0] はマッチした文字列全体、m[1] は 1 番目の部分式、m[2] は 2 番目などとなる。詳細はヘッダファイルの regmatch_t の宣言を見よ。*eflags* 引数は以下の組み合わせである。

フラグ	意味
REG_NOTBOL	引数 *buf* が行の先頭ではない。
REG_NOTEOL	引数 *buf* が行末で終了していない。
REG_STARTEND	検索する文字列は buf + pmatch[0].rm_so が先頭で、buf + pmatch[0].rm_eo が終端である。

regfree

regfree は regcomp が割り当てたメモリをすべて解放する。

コンセプト

charT の要件

basic_regex テンプレートクラスのテンプレート引数で使用する型 charT は、自明なデフォルトコンストラクタ、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持たなければならない。加えてオブジェクトについては以下の要件を満足しなければならない。以下の表では charT 型の *c*、charT const 型の *c1* および *c2*、int 型の *i* を用いる。

式	戻り値の型	表明、注釈、事前・事後条件
charT c	charT	デフォルトコンストラクタ（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* と *c2* の値が同じであれば真。
c1 != c2	bool	*c1* と *c2* が同値でなければ真。
c1 < c2	bool	*c1* の値が *c2* よりも小さければ真。
c1 > c2	bool	*c1* の値が *c2* よりも大きければ真。
c1 <= c2	bool	*c1* が *c2* 以下であれば真。
c1 >= c2	bool	*c1* が *c2* 以上であれば真。
intmax_t i = c1	int	charT は整数型に変換可能でなければならない。 注意：特性クラスが最小限の標準インターフェイスではなく Boost 固有のフルインターフェイスをサポートする場合は、charT 型はこの操作をサポートする必要はない（後述の特性クラスの要件を見よ）。
CharT c(i);	charT	charT は整数型から構築可能でなければならない。

特性クラスの要件

basic_regex の traits テンプレート引数に対しては要件のセットが 2 つある。最小限のインターフェイス（正規表現標準草案の一部）と、オプションの Boost 固有強化インターフェイスである。

最小限の要件

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を表す。*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* および *I2* は入力イテレータ、*c* は const charT 型の値、*s* は型 X::string_type のオブジェクト、*cs* は型 const X::string_type のオブジェクト、*b* は bool 型の値、*I* は int 型の値、*F1* および *F2* は const charT* 型の値、*loc* は X::locale_type 型のオブジェクトである。

式	戻り値の型	表明、注釈、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::size_type	—	charT の null 終端文字列の長さを保持可能な符号なし整数型。

式	戻り値の型	表明、注釈、事前・事後条件
`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::length(p)`	`X::size_type`	`p[i] == 0` である最小の *i* を返す。計算量は *i* に対して線形である。
`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場合に *c* と等価、つまり `v.translate_nocase(c) == v.translate_nocase(C)` となるような文字 *C* を返す。
`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::sting_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。[9]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（あれば）。
`v.getloc()`	`X::locale_type`	*v* が使用中のロカールを返す（あれば）。

9 *I* の値は 8、10、16 のいずれかである。

オプションの追加要件

以下の追加要件は厳密にはオプションである。しかしながら basic_regex でこれらの追加インターフェイスを利用するには、以下の要件をすべて満たさなければならない。basic_regex はメンバ boost_extensions_tag の有無を検出し、自身を適切に構成する。

式	結果	表明、注釈、事前・事後条件
X::boost_extensions_tag	型の指定はない。	与えられている場合、この表にある拡張がすべて与えられていなければならない。
v.syntax_type(c)	regex_constants::syntax_type	正規表現文法における文字 *c* の意味を表す regex_constants::syntax_type 型のシンボル値を返す。
v.escape_syntax_type(c)	regex_constants::escape_syntax_type	正規表現文法において、*c* の前にエスケープ文字がある場合（式中で文字 *c* の直前に文字 *b* がある場合 v.syntax_type(b) == syntax_escape）の文字 *c* の意味を表す regex_constants::escape_syntax_type 型のシンボル値を返す。
v.translate(c, b)	X::char_type	*c* と等価、つまり v.translate(c, false) == v.translate(d, false) となる文字 *d* を返す。あるいは大文字小文字を区別せずに比較した場合に等価、つまり v.translate(c, true) == v.translate(C, true) となる文字 *C* を返す。
v.toi(I1, I2, I)	charT か int を保持可能な整数型。	I1 == I2 か *I1 が数字でなければ-1 を返す。それ以外の場合はシーケンス [I1, I2) に入力数値書式化処理を行い、結果を int で返す。事後条件：I1 == I2 か *I1 が数字以外のいずれか。
v.error_string(I)	std::string	エラー状態 *I* の可読性の高いエラー文字列を返す。*I* は regex_constants::error_type 型が列挙する値のいずれかである。値 *I* が解釈不能な場合は、文字列 “Unknown error” か同じ意味の地域化文字列を返す。
v.tolower(c)	X::char_type	*c* を小文字に変換する。Perl スタイルの \l および \L 書式化処理で使用する。
v.toupper(c)	X::char_type	*c* を大文字に変換する。Perl スタイルの \u および \U 書式化処理で使用する。

イテレータの要件

正規表現アルゴリズム（およびイテレータ）は、すべて双方向イテレータの要件を満たす。

非推奨のインターフェイス

（本節の翻訳は割愛します。内容については原文を参照してください。）

内部の詳細

Unicode イテレータ

概要

```
#include <boost/regex/pending/unicode_iterator.hpp>
```

```
template <class BaseIterator, class U16Type = ::boost::uint16_t>
class u32_to_u16_iterator;

template <class BaseIterator, class U32Type = ::boost::uint32_t>
class u16_to_u32_iterator;

template <class BaseIterator, class U8Type = ::boost::uint8_t>
class u32_to_u8_iterator;

template <class BaseIterator, class U32Type = ::boost::uint32_t>
class u8_to_u32_iterator;

template <class BaseIterator>
class utf16_output_iterator;

template <class BaseIterator>
class utf8_output_iterator;
```

説明

このヘッダに含まれるのは、あるエンコーディングの文字シーケンスを別のエンコーディングの読み取り専用文字シーケンス「のように見せる」イテレータアダプタ群である。

```
template <class BaseIterator, class U16Type = ::boost::uint16_t>
class u32_to_u16_iterator
{
   u32_to_u16_iterator();
   u32_to_u16_iterator(BaseIterator start_position);

   // 他の標準双方向イテレータのメンバが続く...
};
```

UTF-32 文字シーケンスを（読み取り専用の）UTF-16 文字シーケンスに変換する双方向イテレータアダプタである。UTF-16 文字はプラットフォーム標準のバイト順で符号化する。

```
template <class BaseIterator, class U32Type = ::boost::uint32_t>
class u16_to_u32_iterator
{
   u16_to_u32_iterator();
   u16_to_u32_iterator(BaseIterator start_position);
```

```
   u16_to_u32_iterator(BaseIterator start_position, BaseIterator start_range, BaseIterator
end_range);

   // 他の標準双方向イテレータのメンバが続く...
};
```

UTF-16 文字シーケンス（バイト順はプラットフォーム標準）を（読み取り専用の）UTF-32 文字シーケンスに変換する双方向イテレータアダプタである。

このクラスの 3 引数のコンストラクタは、元のシーケンスの開始・終了とともに走査開始位置を取る。このコンストラクタは元のシーケンスの終端が正しく符号化されているか確認する。これにより、元の範囲の終端に不正な UTF-16 コードシーケンスがあった場合にシーケンスの終端を越えて誤って前進・後退するのを防止する。

```
template <class BaseIterator, class U8Type = ::boost::uint8_t>
class u32_to_u8_iterator
{
   u32_to_u8_iterator();
   u32_to_u8_iterator(BaseIterator start_position);

   // 他の標準双方向イテレータのメンバが続く...
};
```

UTF-32 文字シーケンスを（読み取り専用の）UTF-8 文字シーケンスに変換する双方向イテレータアダプタである。

```
template <class BaseIterator, class U32Type = ::boost::uint32_t>
class u8_to_u32_iterator
{
   u8_to_u32_iterator();
   u8_to_u32_iterator(BaseIterator start_position);
   u8_to_u32_iterator(BaseIterator start_position, BaseIterator start_range, BaseIterator
end_range);

   // 他の標準双方向イテレータのメンバが続く...
};
```

UTF-8 文字シーケンスを（読み取り専用の）UTF-32 文字シーケンスに変換する双方向イテレータアダプタである。

このクラスの 3 引数のコンストラクタは、元のシーケンスの開始・終了とともに走査開始位置を取る。このコンストラクタは元のシーケンスの終端が正しく符号化されているか確認する。これにより、元の範囲の終端に不正な UTF-8 コードシーケンスがあった場合にシーケンスの終端を越えて誤って前進・後退するのを防止する。

```
template <class BaseIterator>
class utf16_output_iterator
{
   utf16_output_iterator(const BaseIterator& b);
   utf16_output_iterator(const utf16_output_iterator& that);
   utf16_output_iterator& operator=(const utf16_output_iterator& that);

   // 他の標準出力イテレータのメンバが続く...
};
```

UTF-32 値を入力として受け取り、*BaseIterator b* に UTF-16 で出力する単純な出力イテレータアダプタである。UTF-32 および UTF-16 値はプラットフォーム標準のバイト順でなければならない。

```
template <class BaseIterator>
class utf8_output_iterator
{
   utf8_output_iterator(const BaseIterator& b);
   utf8_output_iterator(const utf8_output_iterator& that);
   utf8_output_iterator& operator=(const utf8_output_iterator& that);

   // 他の標準出力イテレータのメンバが続く...
};
```

UTF-32 値を入力として受け取り、*BaseIterator b* に UTF-8 で出力する単純な出力イテレータアダプタである。UTF-32 入力値はプラットフォーム標準のバイト順でなければならない。

様々な背景に関する情報

ヘッダ

このライブラリで使用する主なヘッダは 2 つある。<boost/regex.hpp>が主要なテンプレートライブラリへの完全なアクセスを提供するのに対し、<boost/cregex.hpp>は（非推奨の）高水準クラス RegEx と POSIX API 関数へのアクセスを提供する。

インターフェイスが basic_regex に依存するものの他に完全な定義が必要ない場合に使用する、前方宣言だけが入ったヘッダ<boost/regex_fwd.hpp>もある。

地域化

Boost.Regex は実行時の地域化について広範のサポートを提供する。この地域化モデルは、フロントエンドとバックエンドの 2 つの部分に分けられる。

フロントエンドの地域化は、エラーメッセージや正規表現構文そのものといったユーザが実際に触れるすべてのものに深く関わる。例えばフランス語のアプリケーションは[[:word:]]を[[:mot:]]に、\wを\mに変更できる。フロントエンドのロカールを変更するには、地域化済み文字列を含んだメッセージカタログを開発者が提供しなければならない。フロントエンドのロカールは LC_MESSAGES カテゴリのみの影響を受ける。

バックエンドの地域化は、正規表現を解析した後に起こるすべてのこと、言い換えるとユーザが直接触れないすべてのものに深く関わる。大文字小文字の変換、照合、文字クラスのメンバーシップがそうである。バックエンドのロカールは開発者の介在を要求しない。ライブラリは現在のロカールについて必要なすべての情報を、オペレーティングシステムや実行時ライブラリから得る。これは例えば正規表現が C++プログラムに組み込まれている場合など、プログラムのユーザが正規表現に直接触れない場合、ライブラリがすべてを取り計らうので明示的な地域化は不要ということを意味する。例えばコードに組み込まれた正規表現[[:word:]]+は常に単語全体にマッチし、プログラムが例えばギリシャ語ロカールのマシンで走っている場合は、ラテン文字ではなくギリシャ文字の単語全体にマッチする。バックエンドのロカールは LC_TYPE および LC_COLLATE カテゴリの影響を受ける。

個別の地域化の機構が 3 つ、Boost.Regex によりサポートされている。

Win32 地域化モデル

ライブラリを Win32 のもとでコンパイルした場合の既定で、特性クラス w32_regex_traits によりカプセル化される。このモデルを使用する場合、basic_regex オブジェクトは各自で LCID を保持する。既定ではこれは GetUserDefaultLCID が返すユーザ既定の設定だが、必要な場合は basic_regex オブジェクトの imbue を呼び出して他の LCID を設定することも可能である。Boost.Regex が使用する設定はすべて C 実行時ライブラリ経由で直接オペレーティングシステムから得る。フロントエンドの地域化では、ユーザ定義文字列の入ったストリングテーブルを含むリソース DLL が必要である。特性クラスは関数

```
static std::string set_message_catalogue(const std::string& s);
```

をエクスポートし、あらゆる正規表現をコンパイルする前（basic_regex インスタンスを構築する前である必要はない）にリソース

DLL の名前を識別する文字列とともに呼び出す必要がある。

```
boost::w32_regex_traits<char>::set_message_catalogue("mydll.dll");
```

NT のもとではライブラリは Unicode を完全にサポートする。9x においては限定的であり、0 から 255 までの文字はサポートするが残りは「不明な」図形文字として扱う。

C 地域化モデル

C++ロカールがあるので、C++ロカールをサポートする非 Win32 コンパイラではこのモデルは非推奨である。このロカールは特性クラス c_regex_traits によりカプセル化され、Win32 ユーザはプリプロセッサシンボル BOOST_REGEX_USE_C_LOCALE を定義してこのモデルを有効化できる。このモデルが有効な場合、setlocale で設定可能な大域ロカールが 1 つだけ存在することになる。すべての設定は実行時ライブラリから得るため、したがって Unicode サポートは実行時ライブラリの実装による。

フロントエンドの地域化はサポートしない。

setlocale を呼び出すとコンパイル済みの正規表現がすべて無効になることに注意していただきたい。setlocale(LC_ALL, "C") を呼び出すと、このライブラリの動作は大部分の旧式の正規表現ライブラリ（本ライブラリのバージョン 1 含む）と同じになる。

C++地域化モデル

Windows 以外のコンパイラではこのモデルが既定である。

このモデルが有効な場合、basic_regex の各インスタンスは自身の std::locale を持つ。また、basic_regex クラスは正規表現のインスタンスごとにロカールを設定するメンバ関数 imbue を持つ。フロントエンドの地域化には POSIX メッセージカタログが必要であり、正規表現が使用するロカールの std::messages ファセットにより読み込まれる。特性クラスは次のシンボルをエクスポートし、

```
static std::string set_message_catalogue(const std::string& s);
```

メッセージカタログの名前を識別する文字列を使って、あらゆる正規表現をコンパイルする前に呼び出す必要がある（が、basic_regex インスタンスを構築する前である必要はない）。

```
boost::cpp_regex_traits<char>::set_message_catalogue("mycatalogue");
```

basic_regex<>::imbue を呼び出すと、その basic_regex インスタンスの正規表現が無効になることに注意していただきたい。

ライブラリを既定以外の地域化モデルでビルドした場合、サポートライブラリをビルドするときと、<boost/regex.hpp>か<boost/cregex.hpp>をインクルードするときの両方で、適切なプリプロセッサシンボル（BOOST_REGEX_USE_C_LOCALE か BOOST_REGEX_USE_CPP_LOCALE）を定義しなければならない。この場合は<boost/regex/user.hpp>に#define を追加するのが最適である。

メッセージカタログの提供

ライブラリのフロントエンドを地域化するためには、リソース DLL のストリングテーブル（Win32 モデル）か POSIX メッセージカタロ

グ（C++モデル）に適切なメッセージ文字列を含めたライブラリを提供する必要がある。後者の場合、カタログのメッセージセット 0 にメッセージを入れておかなければならない。メッセージとその ID は以下のとおりである。

メッセージ ID	意味	既定値
101	部分式の開始に使用する文字。	"("
102	部分式の終了宣言に使用する文字。	")"
103	行末表明の表現に使用する文字。	"$"
104	行頭表明の表現に使用する文字。	"^"
105	「あらゆる文字にマッチする正規表現」の表現に使用する文字。	"."
106	0 回以上の繰り返しにマッチする演算子。	"*"
107	1 回以上の繰り返しにマッチする演算子。	"+"
108	0 回か 1 回の繰り返しにマッチする演算子。	"?"
109	文字集合開始文字。	"["
110	文字集合終了文字。	"]"
111	選択演算子。	"|"
112	エスケープ文字。	"\"
113	ハッシュ文字（現在未使用）。	"#"
114	範囲演算子。	"-"
115	繰り返し演算子開始文字。	"{"
116	繰り返し演算子終了文字。	"}"
117	数字。	"0123456789"
118	エスケープ文字の直後に置いて単語境界表明を表現する文字。	"b"
119	エスケープ文字の直後に置いて非単語境界表明を表現する文字。	"B"
120	エスケープ文字の直後に置いて単語先頭表明を表現する文字。	"<"
121	エスケープ文字の直後に置いて単語終端表明を表現する文字。	">"
122	エスケープ文字の直後に置いて単語構成文字を表現する文字。	"w"
123	エスケープ文字の直後に置いて非単語構成文字を表現する文字。	"W"
124	エスケープ文字の直後に置いてバッファ先頭表明を表現する文字。	"`A"
125	エスケープ文字の直後に置いてバッファ終端表明を表現する文字。	"'z"
126	改行文字。	"\n"
127	カンマ演算子。	","
128	エスケープ文字の直後に置いてベル文字を表現する文字。	"a"
129	エスケープ文字の直後に置いてフォームフィード文字を表現する文字。	"f"
130	エスケープ文字の直後に置いて改行文字を表現する文字。	"n"
131	エスケープ文字の直後に置いて復改文字を表現する文字。	"r"
132	エスケープ文字の直後に置いてタブ文字を表現する文字。	"t"
133	エスケープ文字の直後に置いて垂直タブ文字を表現する文字。	"v"

メッセージ ID	意味	既定値
134	エスケープ文字の直後に置いて 16 進定数を表現する文字。	"x"
135	エスケープ文字の直後に置いて ASCII エスケープ文字の開始を表現する文字。	"c"
136	コロン文字。	":"
137	イコール文字。	"="
138	エスケープ文字の直後に置いて ASCII エスケープ文字を表現する文字。	"e"
139	エスケープ文字の直後に置いて小文字を表現する文字。	"l"
140	エスケープ文字の直後に置いて非小文字を表現する文字。	"L"
141	エスケープ文字の直後に置いて大文字を表現する文字。	"u"
142	エスケープ文字の直後に置いて非大文字を表現する文字。	"U"
143	エスケープ文字の直後に置いて空白類文字を表現する文字。	"s"
144	エスケープ文字の直後に置いて非空白類文字を表現する文字。	"S"
145	エスケープ文字の直後に置いて 10 進数字を表現する文字。	"d"
146	エスケープ文字の直後に置いて非 10 進数字を表現する文字。	"D"
147	エスケープ文字の直後に置いて引用終了演算子を表現する文字。	"E"
148	エスケープ文字の直後に置いて引用開始演算子を表現する文字。	"Q"
149	エスケープ文字の直後に置いて Unicode 結合文字シーケンスを表現する文字。	"X"
150	エスケープ文字の直後に置いて単一文字を表現する文字。	"C"
151	エスケープ文字の直後に置いてバッファ終端演算子を表現する文字。	"Z"
152	エスケープ文字の直後に置いて継続表明を表現する文字。	"G"
153	(?の直後に置いてゼロ幅否定前方先読み表明を表現する文字。	"!"

カスタムのエラーメッセージは以下のように読み込まれる。

メッセージ ID	エラーメッセージ ID	既定の文字列
201	REG_NOMATCH	"No match"
202	REG_BADPAT	"Invalid regular expression"
203	REG_ECOLLATE	"Invalid collation character"
204	REG_ECTYPE	"Invalid character class name"
205	REG_EESCAPE	"Trailing backslash"
206	REG_ESUBREG	"Invalid back reference"
207	REG_EBRACK	"Unmatched [or \["
208	REG_EPAREN	"Unmatched (or \("
209	REG_EBRACE	"Unmatched \{"
210	REG_BADBR	"Invalid content of \{\}"
211	REG_ERANGE	"Invalid range end"
212	REG_ESPACE	"Memory exhausted"
213	REG_BADRPT	"Invalid preceding regular expression"

メッセージ ID	エラーメッセージ ID	既定の文字列
214	REG_EEND	"Premature end of regular expression"
215	REG_ESIZE	"Regular expression too big"
216	REG_ERPAREN	"Unmatched) or \)"
217	REG_EMPTY	"Empty expression"
218	REG_E_UNKNOWN	"Unknown error"

カスタムの文字クラス名は以下のように読み込まれる。

メッセージ ID	説明	等価な既定クラス名
300	アルファベット文字と数字の文字クラス名。	"alnum"
301	アルファベット文字の文字クラス名。	"alpha"
302	制御文字の文字クラス名。	"cntrl"
303	10 進数字の文字クラス名。	"digit"
304	図形文字の文字クラス名。	"graph"
305	小文字の文字クラス名。	"lower"
306	印字可能文字の文字クラス名。	"print"
307	区切り文字の文字クラス名。	"punct"
308	空白の文字クラス名。	"space"
309	大文字の文字クラス名。	"upper"
310	16 進数字の文字クラス名。	"xdigit"
311	行区切り以外の空白類文字の文字クラス名。	"blank"
312	単語構成の文字クラス名。	"word"
313	Unicode 文字の文字クラス名。	"unicode"

最後にカスタムの照合要素名はメッセージ ID 400 から読み込まれ、最初に失敗したところで終了する。各メッセージは “tagname string” のような形式で、*tagname* は`[[.tagname.]]`の内部で使用する名前、*string* は照合要素の実際のテキストである。照合要素`[[.zero.]]`の値は文字列から数値への変換に使用され、他の値で置換するとその値が文字列解析に使われるということに注意していただきたい。例えば正規表現内でラテン数字の代わりに Unicode のアラビア-インド数字を使用するのであれば、`[[.zero.]]`に Unicode 文字 0x0660 を充てればよい。

カスタム名を定義した場合でも、文字クラスおよび照合要素の POSIX 定義名は常に有効であるということに注意していただきたい。一方、カスタムのエラーメッセージとカスタムの構文メッセージは既存のものを上書きする。

スレッド安全性

Boost がスレッド安全であれば、Boost.Regex はスレッド安全である。Boost がスレッド安全モードであるかどうか確認するには、`BOOST_HAS_THREADS` が定義されているか調べるとよい。コンパイラがスレッドのサポートを有効にしていれば、設定システムがこのマクロを自動的に定義する。

`basic_regex` クラスとその typedef である `regex`、`wregex` は、コンパイル済み正規表現がスレッド間で安全に共有可能という意

味でスレッド安全である。マッチアルゴリズム regex_match、regex_search および regex_replace はすべて再入可能かつスレッド安全である。match_results クラスは、マッチ結果をあるスレッドから別のスレッドへ安全にコピー（例えばあるスレッドがマッチを検索して match_results インスタンスをキューに追加し、同時に別のスレッドが同じキューをポップすることが）可能という意味では、スレッド安全である。それ以外の場合はスレッドごとに個別の match_results インスタンスを使用しなければならない。

POSIX API 関数はすべて再入可能かつスレッド安全であり、regcomp でコンパイルした正規表現もスレッド間で共有可能である。

RegEx クラスは、各スレッドが RegEx インスタンスを保持する場合のみスレッド安全である（アパートメントスレッディング）。これは RegEx が正規表現のコンパイルとマッチの両方を処理するためである。

最後に、大域ロカールを変更するとあらゆるコンパイル済み正規表現が無効になるため、あるスレッドで正規表現を使用しているときに別のスレッドが setlocale を呼び出すと予期しない結果となることに注意していただきたい。

また main() の開始前は、実行中のスレッドは 1 つだけでなければならないという要件がある。

テストとサンプルプログラム

テストプログラム

regress:

退行テストアプリケーションはマッチ・検索アルゴリズムを完全にテストする。このプログラムが存在することにより、ライブラリが要求どおりに動作する（少なくともテストにある項目はテストされている）ことが保証される。未テストの項目を発見された方がおられたら、よろこんで拝聴するしだいである。

ファイル：

- main.cpp
- basic_tests.cpp
- test_alt.cpp
- test_anchors.cpp
- test_asserts.cpp
- test_backrefs.cpp
- test_deprecated.cpp
- test_emacs.cpp
- test_escapes.cpp
- test_grep.cpp
- test_icu.cpp
- test_locale.cpp
- test_mfc.cpp
- test_non_greedy_repeats.cpp
- test_operators.cpp
- test_overloads.cpp

- test_perl_ex.cpp
- test_replace.cpp
- test_sets.cpp
- test_simple_repeats.cpp
- test_tricky_cases.cpp
- test_unicode.cpp

bad_expression_test:

「不正な」正規表現により無限ループが発生せず、例外が投げられることを検証する。

ファイル：bad_expression_test.cpp

recursion_test:

（正規表現が何であるかに関わらず）スタックオーバーランを起こさないことを検証する。

ファイル：recursion_test.cpp

concepts:

ライブラリがドキュメントにあるコンセプトをすべて満たしているか検証する（コンパイルのみのテスト）。

ファイル：concept_check.cpp

captures_test:

捕捉をテストするコード。

ファイル：captures_test.cpp

サンプルプログラム

grep

簡単な grep の実装。-h コマンドラインオプションを付けて走らせると使用法が表示される。

ファイル：grep.cpp

timer.exe

簡単な対話式の正規表現マッチアプリケーション。結果はすべて計時される。効率が問題となる場合に、プログラマはこのプログラムを使って正規表現の最適化を行うことができる。

ファイル：regex_timer.cpp

コード片

コード片の例は本ドキュメントで使用したコード例である。

captures_example.cpp：捕捉のデモンストレーション。

credit_card_example.cpp：クレジットカード番号の書式化コード。

partial_regex_grep.cpp：部分マッチを使った検索の例。

partial_regex_match.cpp：regex_match で部分マッチを使った例。

regex_iterator_example.cpp：マッチの一連を反復する。

regex_match_example.cpp：FTP を題材にした regex_match の例。

regex_merge_example.cpp：regex_merge の例。C++ファイルを、構文強調した HTML に変換する。[10]

regex_replace_example.cpp：regex_replace の例。C++ファイルを、構文強調した HTML に変換する。

regex_search_example.cpp：regex_search の例。cpp ファイルからクラス定義を検索する。

regex_token_iterator_eg_1.cpp：文字列をトークン列に分割する。

regex_token_iterator_eg_2.cpp：HTML ファイル内の URL リンクを列挙する。

以下は非推奨である。

（非推奨の例については省略させていただきます。）

参考文献とさらなる情報

正規表現の短いチュートリアルがこことここにある。

正規表現に関する最も重要な書籍は O'Reilly 出版の Mastering Regular Expressions である。[11]

Boost.Regex は、Technical Report on C++ Library Extensions の正規表現に関する章の基本部分をなす。

Open Unix Specification には POSIX 正規表現構文などの有益な資料が豊富にある。

Pattern Matching Pointers はパターンマッチに興味のある人には「必見の」サイトである。

Glimpse と Agrep は簡略化した正規表現構文を使用して検索時間の高速化を実現している。

Udi Manber と Ricardo Baeza-Yates のサイトには、いずれもパターンマッチに関する有益な論文へのリンクがある。

よくある質問と回答

Q. Boost.Regex でエスケープ文字がうまく動作しないが、何がまずいのか。

A. C++のコードに正規表現を埋め込む場合は、エスケープ文字が 2 度処理されることを忘れてはならない。1 度目は C++コンパイラで 2 度目は Boost.Regex の正規表現コンパイラである。よって正規表現`\d+`を Boost.Regex に渡すときは`"\\d+"`をコードに埋め込む必要がある。同様に直値のバックスラッシュをマッチさせるには、コードには`"\\"`を埋め込む必要がある。

Q. どうやっても regex_match が常に偽を返すのだが、何がまずいのか。

A. アルゴリズム regex_match は、正規表現のマッチがテキスト**全体**に対して成功する場合のみ成功する。テキスト内の、正規表現にマッチする部分文字列を**検索**する場合は regex_search を使用する。

Q. POSIX 正規表現で括弧を使うとマッチ結果が変わるのはなぜか。

A. Perl と異なり、POSIX（拡張および基本）正規表現では、括弧はマーク付けを行うだけでなく最良マッチの決定も行う。正規表現を POSIX 基本、あるいは POSIX 拡張正規表現としてコンパイルすると、Boost.Regex は POSIX 標準の最左最長規則にしたがってマッチを決定する。よって正規表現全体を評価した上で可能性のあるマッチが 2 つ以上ある場合、次の 1 番目の部分式を、その次は 2 番目の部分式、となる。そのため、`"00123"` に対して`(0*)([0-9]*)`でマッチをかけると$1 = "00"、$2 = "123" が生成されるが、`"00123"` に対して`0*([0-9]*)`でマッチをかけると$1 = "00123" が生成される。

$1 を `"123"` の部分のみにマッチさせたい場合は、`0*([1-9][0-9]*)`といった正規表現を使用するとよい。

10 訳注 regex_merge は非推奨機能の 1 つです。本文書（日本語訳）には記述はありません。

11 訳注 邦訳は『詳説 正規表現 第 3 版』、長尾高弘 訳、オライリー・ジャパン、2008 年。

Q. 文字範囲が正しく動作しないのはなぜか（POSIX モードのみ）。

A. POSIX 標準は文字範囲式がロカールを考慮すると規定している。よって、例えば正規表現 `[A-Z]` は照合順が ‘A’ から ‘Z’ の間となるあらゆる照合要素にマッチする。これが意味するところは “C” と “POSIX” 以外の大部分のロカールでは、`[A-Z]` が例えば "ヒ" 1 文字にマッチするという、大部分の人が期待しない（あるいは少なくとも大部分の人が正規表現エンジンの動作として考えていない）動作をするということである。このような理由から、Boost.Regex の既定の動作（Perl モード）ではコンパイルフラグ `regex_constants::collate` を設定しない限りロカール依存の照合は無効になっている。しかしながら、`regex_constants::extended` や `regex_constants::basic` といった既定以外のコンパイルフラグを設定するとロカール依存の照合が有効になる。これは内部的に `regex_constants::extended` や `regex_constants::basic` を使用する POSIX API 関数にも適用される。[12]

Q. 関数に例外仕様がないのはなぜか。ライブラリが投げる例外にはどのようなものがあるか。

A. すべてのコンパイラが例外仕様をサポート（あるいは尊重）しているわけではなく、サポートしているコンパイラの中にも効率が低下するものがある。コンパイラの処理が現在よりも改善される日が来れば、例外仕様を追加するかもしれない。本ライブラリが投げる例外は 3 種類のみである。`boost::regex_error` は basic_regex が正規表現のコンパイル中に投げる可能性がある。`std::runtime_error` は、`basic_regex::imbue` が存在しないメッセージカタログを開こうとしたとき、regex_search および regex_match の検索時間が「限界を超えた」とき、あるいは `RegEx::GrepFiles` および `RegEx::FindFiles` が開けないファイルを開こうとしたときに投げる。最後に `std::bad_alloc` は本ライブラリのあらゆる関数が投げる可能性がある。

Q. `regex_match`、`regex_search`、`regex_grep`、`regex_format`、`regex_merge` の「便利」版が使用できないのはなぜか。

A. これらの版が利用可能かどうかはコンパイラの能力に依存する。これらの関数の形式を決定する規則はかなり複雑であり、ヘルプに掲載しているのは標準準拠コンパイラでの形式のみである。あなたのコンパイラがどの形式をサポートしているか調べるには、<boost/regex.hpp>を C++プリプロセッサにかけ、出力ファイルから目的の関数を探すとよい。しかしながら、ごく少数のコンパイラがこれらの関数の多重定義を正しく解釈できないことに注意していただきたい。

効率

Boost.Regex の再帰モードと非再帰モード両方の効率については、他の幅広い正規表現ライブラリと比較されてしかるべきである。再帰モードは少しばかり高速（主にメモリ割り当てにスレッドの同期が必要な場合）だが、あまり大きな差は無い。以下のページで 2 種類のコンパイラを用いて数種類の正規表現ライブラリと比較を行っている。

- Visual Studio .Net 2003（再帰的な Boost.Regex 実装）
- GCC 3.2（Cygwin）（非再帰的な Boost.Regex 実装）

12 `regex_constants::nocollate` が有効な場合は、ライブラリの動作は他の設定に関わらず `LC_COLLATE` ロカールカテゴリが “C” であるのと同様になる。

標準への適合

C++

Boost.Regex は Technical Report on C++ Library Extensions への適合を意図している。

ECMAScript / JavaScript

以下を除く ECMAScript 正規表現構文のすべての機能をサポートする。

エスケープシーケンス`\u`は Unicode エスケープシーケンスではなく、大文字にマッチする（`[[:upper:]]`と同じ）。Unicode エスケープには`\x{DDDD}`を使用する。

Perl

以下を除く Perl のほとんどすべての機能をサポートする。

- `(?{code})`はコンパイルの必要な型付けの強い言語では実装不可能である。
- `(??{code})`はコンパイルの必要な型付けの強い言語では実装不可能である。
- 特殊なバックトラック制御記号（`(*VERB)`）[13]は、現時点では実装していない。
- また、`^` `$` `\Z`の機能は Perl とは少し動作が異なる。これらは`\n`以外の改行シーケンスも取り扱う。後述の Unicode の要件を見よ。

POSIX

以下を除く POSIX 標準および拡張のすべての機能をサポートする。

C ロカールでは特性クラスを使って明示的に登録しない限り、POSIX 標準で指定されているもの以外の照合名は解釈されない。

文字等価クラス（`[[=a=]]`など）は Win32 以外ではおそらくバグがある。この機能の実装には、システムが生成する文字列ソートキーについての情報が必要である。この機能が必要で、既定の実装が使用しているプラットフォームで動作しない場合は、カスタムの特性クラスを用意する必要がある。

Unicode

以下のコメントは Unicode Technical Report #18: Unicode Regular Expressions version 11 に対するものである。

項目	機能	サポート
1.1	16 進表記 Hex Notation	サポート。コードポイント U+DDDD を表すには`\x{DDDD}`を使用する。
1.2	文字プロパティ Character Properties	一般分類の名前はすべてサポートする。用字系およびその他の名前は現時点ではサポートしない。
1.3	差および交差	前方先読みにより間接的にサポートする。

13 訳注 “Special backtracking control verbs”。Perl の新しい実験的機能のため、定訳らしきものがないのかもしれません。例えば http://fleur.hio.jp/perldoc/perl/5.9.5/pod/perlre.ja.html を参照してください。

項目	機能	サポート
	Subtractions and Intersections	`(?=[[:X:]])[[:Y:]]`は文字プロパティ*X*と*Y*の交差を与える。 `(?![[:X:]])[[:Y:]]`は*Y*の中で*X*に含まれないもの（差）を与える。
1.4	単純な単語境界 Simple Word Boundaries	適合する。送りなし記号は単語構成文字として扱う。
1.5	大文字小文字を区別しないマッチ Caseless Matching	サポートする。この水準では大文字小文字の対応は一対一の変換であり、多対多のケースフォールディング（“ß” から “SS” への変換）はサポートしない。
1.6	行境界 Line Boundaries	`.`が `"\r\n"` の 1 文字ずつにしかマッチしないということ以外はサポートする。単語境界以外は正しくマッチする（`"\r\n"` シーケンスの中間にマッチしない）。
1.7	コードポイント Code Points	サポートする。`u32*`アルゴリズムを使用して UTF-8、UTF-16 および UTF-32 をすべて 32 ビットコードポイント列として扱うことができる。
2.1	正規等価 Canonical Equivalence	サポートしない。ライブラリのユーザがテキストを正規表現と同じ正規形に変換するしかない。
2.2	既定の書記素 Default Grapheme Clusters	サポートしない。
2.3	既定の単語境界 Default Word Boundaries	サポートしない。
2.4	既定のあいまいマッチ Default Loose Match	サポートしない。
2.5	名前付きプロパティ Named Properties	サポートする。正規表現`[[:name:]]`、`\N{Name}`は名前付き文字 “name” にマッチする。
2.6	プロパティ名中のワイルドカード Wildcard Properties	サポートしない。
3.1	区切り文字のテーラリング Tailored Punctuation	サポートしない。
3.2	書記素のテーラリング Tailored Grapheme Clusters	サポートしない。
3.3	単語境界のテーラリング Tailored Word Boundaries	サポートしない。
3.4	テーラリングを用いたあいまいなマッチ Tailored Loose Match	部分的にサポートする。`[[=c=]]`は*c*と同じ第 1 位の等価クラスを持つ文字にマッチする。
3.5	範囲のテーラリング Tailored Ranges	サポートする。`collate`フラグを設定して式を構築した場合、`[a-b]`は a から b の範囲に照合される文字にマッチする。
3.6	文脈を考慮したマッチ Context Matches	サポートしない。
3.7	インクリメンタルマッチ Incremental Matches	サポートする。正規表現アルゴリズムにフラグ`match_partial`を渡す。
3.8	コンパイル済み文字集合の共有 Unicode Set Sharing	サポートしない。

項目	機能	サポート
3.9	可能なマッチの集合 Possible Match Sets	サポートしない。しかしながらこの情報は正規表現のマッチを最適化するために内部的に使用し、可能なマッチが存在しない場合に高速に処理を返すようになっている。
3.10	フォールディング付きのマッチ Folded Matching	サポートしない。カスタムの正規表現特性クラスを用いることで似たような効果を得ることができる。
3.11	カスタムマッチ Custom Submatch Evaluation	サポートしない。

再配布について

Microsoft か Borland の C++を使って実行時ライブラリの DLL 版にリンクしているのであれば、コードをコンパイルするときにシンボル `BOOST_REGEX_DYN_LINK` を定義して Boost.Regex の DLL 版にリンク可能である。これらの DLL は再配布可能だが「標準の」版というものが存在しないので、ユーザの PC にインストールする場合は、PC のディレクトリパスではなくアプリケーションの私的なディレクトリに配置するべきである。実行時ライブラリの静的版にリンクしているのであれば Boost.Regex の静的版にリンクすればよく、DLL の再配布は必要ない。Boost.Regex の DLL、ライブラリがとり得る名前は Getting Started ガイドに与えられている式から決定する。

コンパイル時にシンボル `BOOST_REGEX_NO_LIB` を定義すると、ライブラリの自動選択を無効にできるということに注意していただきたい。Boost.Regex を自分で IDE を使ってビルドしたい場合や Boost.Regex をデバッグする必要がある場合に役に立つ。

謝辞

john - at - johnmaddock.co.uk で著者に連絡を取ることができる。本ライブラリのホームページは www.boost.org である。

私の中のアルゴリズムとその効率に対する考え方は、Robert Sedgewick の『Algorithms in C++』[14]、そして Boost の方々によるところが大きい。以上。

Boost.Xpressive と GRETA 正規表現コンポーネントの作者である Eric Niebler とは、長い議論でさまざまな重要なアイデアを共有した。

Roundhouse Consulting, Ltd.の Pete Becker には言語の標準草案について多大な助力をいただいた。

以下の方々に有益なコメントとバグ修正をいただいた。Dave Abrahams、Mike Allison、Edan Ayal、Jayashree Balasubramanian、Jan Bölsche、Beman Dawes、Paul Baxter、David Bergman、David Dennerline、Edward Diener、Peter Dimov、Robert Dunn、Fabio Forno、Tobias Gabrielsson、Rob Gillen、Marc Gregoire、Chris Hecker、Nick Hodapp、Jesse Jones、Martin Jost、Boris Krasnovskiy、Jan Hermelink、Max Leung、Wei-hao Lin、Jens Maurer、Richard Peters、Heiko Schmidt、Jason Shirk、Gerald Slacik、Scobie Smith、Mike Smyth、Alexander Sokolovsky、Hervé Poirier、Michael Raykh、Marc Recht、Scott VanCamp、Bruno Voigt、Alexey Voinov、Jerry Waldorf、Rob Ward、Lealon Watts、John Wismar、Thomas Witt、Yuval Yosef。

14 訳注　邦訳は『アルゴリズム C++』、野下浩平/星守/佐藤創/田口東 訳、近代科学社、1994 年

もしあなたの名前が入っていなかったら（2、3 人だと思う。名前が分からないだけなのだが…）、知らせてほしい。

Henry Spencer、PCRE、Perl および GNU 正規表現ライブラリのマニュアルにも感謝したい。これらのライブラリおよび POSIX 標準との互換性を維持しようと考えたときはいつもこれらのマニュアルが役に立った。しかしながら、コードはバグも含めて（！）私自身によるものである。 知らないバグは修正できるはずもないので、コメントやバグがあれば知らせてほしい。

履歴

新規項目は svn.boost.org に提出してほしい。チケットにはあなたのメールアドレスを入れておく必要がある。

現在オープンな項目はここから見られる。

クローズなものを含めた全項目はここから見られる。

Boost.Regex-5.0.1（Boost-1.58.0）

- 誤字を修正した（#10682）。
- Coverity の警告についてプルリクエスト#6 をマージした。
- Coverity の警告についてプルリクエスト#7 をマージした。
- Coverity の警告についてプルリクエスト#8 をマージした。
- ICU にリンクする場合により多くのビルドバリアントを可能にするためプルリクエスト#10 をマージした。
- ICU と部分マッチを組み合わせたときに発生するバグを修正した（#10114）。
- ICU ライブラリの遅延ロードサポートを削除した。遅延ロードは最新の ICU リリースでは（リンカエラーにより）動作しない。

Boost.Regex-5.0.0（Boost-1.56.0）

- Git へ移行後、ライブラリ固有のバージョン番号を使用することにした。また、破壊的変更が 1 つあったためバージョン 4 からバージョン 5 となった。
- **破壊的変更：**`basic_regex<>::mark_count()` の挙動を既存のドキュメントと一致するよう修正した。同時に `basic_regex<>::subexpression(n)` がマッチするよう変更した。#9227 を見よ。
- チケット#8903 を修正した。
- ドキュメントの誤字を修正した（#9283）。
- 照合コードについて、ロカールが NUL を含む照合文字列を生成した場合に失敗するバグを修正した。#9451 を見よ。
- まれなスレッドの使用方法（静的に初期化されていないミューテックス）に対するパッチを適用した。#9461 を見よ。
- 不正な UTF-8 シーケンスに対するチェック機能を改善した。#9473 を見よ。

Boost-1.54

- 以下のチケットの修正：#8569。

Boost-1.53

- 以下のチケットの修正：#7744、#7644。

Boost-1.51

- 以下のチケットの修正：#589、#7084、#7032、#6346。

Boost-1.50

- `(?!)`が正しい式とならない問題を修正し、正しい条件式の構成要素についてドキュメントを更新した。

Boost-1.48

- 以下のチケットの修正：#698、#5835、#5958、#5736。

Boost 1.47

- 以下のチケットの修正：#5223、#5353、#5363、#5462、#5472、#5504。

Boost 1.44

- 以下のチケットの修正：#4309、#4215、#4212、#4191、#4132、#4123、#4114、#4036、#4020、#3941、#3902、#3890。

Boost 1.42

- 書式化式として文字列だけでなく関数子も受け付けるようにした。
- 例外を投げたときに、より適切な情報を含めてエラー報告を強化した。
- 再帰式を使用した場合の効率が上がり、スタックの使用量が減少した。
- 以下のチケットの修正：#2802、#3425、#3507、#3546、#3631、#3632、#3715、#3718、#3763、#3764。

Boost 1.40

- 名前付き部分式、選択分岐による部分式番号のリセット、再帰正規表現といった Perl 5.10 の構文要素の多くを追加した。

Boost 1.38

- **破壊的な変更：**Perl の正規表現構文で空の正規表現および空の選択を許容するようにした。この変更は Perl との互換性のためのものである。新しい `syntax_option_type` である `no_empty_expressions` が設定されていれば以前の挙動となり、空の式は許可されない。チケット#1081 にもとづいている。
- 書式化文字列において Perl 形式の`${n}`式をサポートした（チケット#2556）。
- 正規表現文字列内の部分式の位置へのアクセスをサポートした（チケット#2269）。
- コンパイラ互換性について修正を行った（チケット#2244、#2514 および#2458）。

Boost 1.34

- 貪欲でない繰り返しと部分マッチが場合によっては正常に動作しないのを修正した。
- 貪欲でない繰り返しが VC++で場合によっては正常に動作しないのを修正した（バグレポート 1515830）。
- `*this` が部分マッチを表しているときに `match_results::position()` が意味のある結果を返すように変更した。
- 改行文字が`|`と同様に扱われるように `grep` および `egrep` オプションを修正した。

Boost 1.33.1

- メイクファイルが壊れていたのを修正した。
- VC7.1 + STLport-4.6.2 で/Zc:wchar_t を使用してビルドできるように設定セットアップを修正した。
- SGI Irix コンパイラが対処できるように、static_mutex.hpp のクラスインラインの宣言を移動した。
- 必要な標準ライブラリの#include を fileiter.hpp、regex_workaround.hpp および cpp_regex_traits.hpp に追加した。
- 貪欲でない繰り返しが奇妙な事情により最大値よりも多く繰り返す場合があったのを修正した。
- デフォルトコンストラクタで構築したオブジェクトが basic_regex<>::empty() で返す値を修正した。
- regex_error の定義を Boost-1.32.0 と後方互換になるように変更した。
- Intel C++ 8.0 未満で外部テンプレートを無効にした。未解決の参照が発生していた。
- gcc で特定のメンバ関数だけがエクスポートされるように extern なテンプレートコードを書き直した。リンク時にデバッグ用コードと非デバッグコードを混ぜたときに奇妙な未解決の参照が発生していた。
- Unicode イテレータのメンバを初期化するようにした。gcc で不要な警告が出なくなった。
- ICU 関連のコードを VC6 と VC7 に移植した。
- STLport のデバッグモードをクリーン化した（？）。
- 後読み表明を固定長さの繰り返しが使えるように、また反復時に後読みが現在の検索範囲の前に（前回のマッチに）遡れるように修正した。
- 前方先読み内の貪欲でない繰り返しに関する奇妙なバグを修正した。
- 文字集合内で文字クラスの否定が使えるようにした。
- [a-z-]を再び正しい正規表現として退行テストを修正した。
- いくつか不正な式を受け付けていたバグを修正した。

Boost 1.33.0

- 式の解析コードを完全に書き直し、特性クラスのサポートを追加した。これにより標準草案に適合した。
- 破壊的な変更：basic_regex コンストラクタに渡す構文オプションを合理化した。既定のオプション（perl）が値 0 となり、どの正規表現構文スタイル（Perl、POSIX 拡張、POSIX 基本など）にどのオプションを適用できるか明確に文書化した。
- 破壊的な変更：POSIX 拡張正規表現および POSIX 基本正規表現が以前よりも厳密に POSIX 標準に従うようになった。
- (?imsx-imsx)構造のサポートを追加した。
- 先読みの式(?<=positive-lookbehind)および(?<!negative-lookbehind)のサポートを追加した。
- 条件式(?(assertion)true-expression|false-expression)のサポートを追加した。
- MFC/ATL 文字列のラッパを追加した。
- Unicode サポートを追加した。ICU を使用している。
- 改行のサポートについて、\f を行区切り（あらゆる文字型で）、\x85 をワイド文字の行区切り（Unicode のみ）として処理するように変更した。
- 置換文字列を Perl や Sed スタイルの書式化文字列ではなく直値として扱う、新しい書式化フラグ format_literal を追加

した。

- エラーの通知を regex_error 型の例外で表現するようになった。以前使用していた型 bad_expression および bad_pattern は regex_error に対する typedef でしかなくなった。regex_error 型は新しい 2、3 のメンバを持つ。code() は文字列ではなくエラーコードを返し、position() は式中のエラーの発生位置を返す。

Boost 1.32.1

- . の境界付き繰り返しの部分マッチに関するバグを修正した。

Boost 1.31.0

- パターンマッチのコードを完全に書き直した。以前よりも 10 倍速くなった。
- ドキュメントを再編成した。
- 正規表現標準草案にないインターフェイスをすべて非推奨とした。
- regex_iterator と regex_token_iterator を追加した。
- Perl スタイルの独立部分式のサポートを追加。
- sub_match クラスに非メンバ演算子を追加した。これにより sub_match の文字列との比較、および文字列への追加による新文字列の生成が可能になった。
- 拡張的な捕捉情報に対する実験的なサポートを追加した。
- マッチフラグの型を（整数でない別の型に）変更した。マッチフラグを match_flag_type ではなく整数としてアルゴリズムに渡そうとするとコンパイルエラーとなるようになった。