# PartII ライブラリ

## Chapter 1 文字列処理

# 1.1 tokenizer

Boost C++ Libraries Programming　文字列切り分け

例えばユーザーの打ち込んだコマンド文字列をパラメータ毎に切り分ける処理。あるいは、表計算ソフトの出力したコンマ文字(,)区切りのテキストデータを読み込む処理。文字列を何かの基準で区切って複数の文字列のリストに変えるという処理は、非常にしばしばプログラム中に登場する文字列処理の代表例です。Boost.Tokenizerライブラリでは、この、様々な基準に応じて文字列の分割を行う機能を提供します。まずはサンプルコードをご覧下さい。

```
#include <iostream>
#include <string>
#include <boost/tokenizer.hpp>
using namespace std;
using namespace boost;

int main()
{
    // サンプル文字列
    string str = "This is a pen.";

    // 空白文字を分割位置として、サンプル文字列strを分割
    char_separator<char>              sep(" ¥t¥n");
    tokenizer< char_separator<char> > tokens( str, sep );

    // 分割結果を表示
    typedef tokenizer< char_separator<char> >::iterator Iter;
    for( Iter it=tokens.begin(); it!=tokens.end(); ++it )
        cout << "TOKEN: " << *it << endl;

    return 0;
}
```

▲Src. 1-1　tokenizer1.cpp

実行結果

```
> tokenizer1
TOKEN: This
TOKEN: is
TOKEN: a
TOKEN: pen.
```

tokenzierクラステンプレートに、分割したい文字列と、分割基準を定めたオブジェクトを渡せば、分割済み文字列(トークンと言います)の列としてアクセスすることができました。Src. 1-1は、スペースやタブや改行があればそこで区切るという基準でトークン分けを行っています。

これを利用すると、こんなデータの解析が簡単にできそうです。

```
名前,住所,電話番号
山田太郎,A県B市,0123-45-6789
佐藤花子,C県D市,02-0202-0202
...
```

▲Src. 1-2　CSV(コンマ区切りテキスト)

,(コンマ)区切りテキスト文書、いわゆるCSVと呼ばれているデータ形式です。XML全盛の今では大分出番が減ってきましたが、それでもまだ随所で利用されている手軽なデータ形式の一つです。さてでは、tokenizerクラステンプレートを使ってCSV解析器を書いてみましょう。区切り文字をコンマにすればいいのですから…

```
while( getline(cin, str) )
{
    char_separator<char>               sep(",");
    tokenizer< char_separator<char> > tokens( str, sep );
    ...;
}
```

▲Src. 1-3　CSVを解析する例1

とすればコンマによる分割が行われます。

しかし、簡単な場合はこれで良いのですが、「データ文字列中にコンマを使いたい」という要求が出た場合にこの方法だとちょっと困ってしまいます。そんな場合は大抵、コンマが出現するデータを引用符で囲って区切りを禁止するか、コンマを¥記号でエスケープするという方法を取ります。Boost.Tokenizerライブラリでは

この用途のための専用の分割基準が用意されています。

```
#include <iostream>
#include <string>
#include <boost/tokenizer.hpp>
using namespace std;
using namespace boost;

int main()
{
    // 各行読み込み
    string str;
    while( getline(cin,str) )
    {
        // コンマ区切り、ただし""と¥によるエスケープを認識する。
        escaped_list_separator<char>              esc;
        tokenizer< escaped_list_separator<char> > tokens( str, esc );

        // 分割結果の表示
        typedef
          tokenizer< escaped_list_separator<char> >::iterator Iter;
        for( Iter it=tokens.begin(); it!=tokens.end(); ++it )
            cout << "[" << *it << "]   ";
        cout << endl;
    }
    return 0;
}
// 入力例(sample.csv)
// aaa,bbb,ccc
// "aaa,bbb",ccc
// aaa¥,bbb,ccc
// aaa¥¥,bbb,ccc
```

▲Src. 1-4 CSVを解析する例2(tokenizer2.cpp)

実行結果

```
> tokenizer2 < sample.csv
[aaa]   [bbb]   [ccc]
[aaa,bbb]   [ccc]
[aaa,bbb]   [ccc]
[aaa¥]   [bbb]   [ccc]
```

多くの場合はこの二種類、char_separatorとescaped_list_separatorのパラメータを変えるだけで対応できると思います。また、これらできあいの分割基準の他にも、自分で分割基準を定めたクラスを作って、tokenizerクラステンプレートに渡すことも可能です。例として、文字列を、'<'で始まり'>'で終わる「タグ」と残りの「テキスト」へとトークン分解してみましょう。

```
#include <iostream>
#include <string>
#include <iterator>
#include <boost/tokenizer.hpp>
using namespace std;
using namespace boost;

struct tag_separator
{
    // 内部変数(もしあれば)をリセットするreset関数と
    // 一つのトークンだけを読み込むoperator()関数の
    // 二つさえ定義されていれば、tokenizerで使うことができます

    void reset() {} // 今回はresetは実装不要

    template<typename Iterator, typename Token>
      bool operator()( Iterator& i, Iterator end, Token& tok )
      {
          tok = Token();    // tokをクリア
          if( i == end )
              return false; // 読めるトークンがなければfalse

          if( *i == '<' )   // タグを取り出す場合
          {
              // 閉じ括弧の手前までiを進めて、tokにコピー
              for( ; i!=end && *i!='>'; ++i )
                  tok += *i;
              if( i == end )
                  return false;

              // 閉じ括弧の分も
              tok += *i++;
              return true;
          }
          else              // タグでない部分を取り出す場合
```

```
        {
            // 開き括弧の手前までiを進めて、tokにコピー
            for( ; i!=end && *i!='<'; ++i )
                tok += *i;
            return true;
        }
    }
};

int main()
{
    // tag_separatorを使って標準入力全体を分割
    cin.unsetf( ios_base::skipws );
    string str( (istream_iterator<char>(cin)), istream_iterator<char>() );
    tokenizer<tag_separator> tokens( str );

    // 結果表示
    typedef tokenizer<tag_separator>::iterator Iter;
    for( Iter it=tokens.begin(); it!=tokens.end(); ++it )
        cout << "TOKEN: " << *it << endl;
    return 0;
}
```

▲Src. 1-5 tokenizer3.cpp

実行結果

```
> tokenizer3
<?xml version="1.0"?>
<document><title>hello</title><body>world</body></document>
^D
TOKEN: <?xml version="1.0"?>
TOKEN:

TOKEN: <document>
TOKEN: <title>
TOKEN: hello
TOKEN: </title>
TOKEN: <body>
TOKEN: world
TOKEN: </body>
TOKEN: </document>
TOKEN:
```

# Reference

Boost C++ Libraries Programming　文字列切り分け

## tokenizer<TokenizerFunc,Iterator,Type> クラステンプレート

| 必要なヘッダファイル | #include <boost/tokenizer.hpp> |
|---|---|
| 名前空間 | boost |

```
template<typename TokenizerFunc = char_delimiters_separator<char>,
         typename Iterator      = std::string::const_iterator,
         typename Type          = std::string>
  class tokenizer
  {
  public:
      tokenizer( Iterator begin, Iterator end,
                 const TokenizerFunc& f = TokenizerFunc() );
      template<typename Container>
        tokenizer( const Container& c,
                   const TokenizerFunc& f = TokenizerFunc() );

      void assign( Iterator begin, Iterator end );
      void assign( Iterator begin, Iterator end,
                   const TokenizerFunc& f );
      template<typename Container>
        void assign( const Container& c );
      template<typename Container>
        void assign( const Container& c, const TokenizerFunc& f );

      typedef typename token_iterator_generator<
        TokenizerFunc,Iterator,Type>::type iterator;
      iterator begin() const;
      iterator end() const;
  };
```

コンストラクタやassignメンバ関数で渡した文字列をトークン分けし、iteratorを通じてアクセスできるようにするクラスです。テンプレート引数のTokenizerFuncには、後述の「TokenizerFunctionコンセプト」に従ってトークン分け方法を定義したクラスを指定します。Iteratorはトークン分けされる文字列を指すイテレータ型を、Typeには分割結果トークンを格納する型をそれぞれ指定します。

```
tokenizer( Iterator begin, Iterator end,
           const TokenizerFunc& f = TokenizerFunc() );
template<typename Container>
  tokenizer( const Container& c,
             const TokenizerFunc& f = TokenizerFunc() );
```

コンストラクタです。トークン分けしたい文字列を、二つのイテレータか、もしくはコンテナの形で渡します。コンテナ版は( c.begin(), c.end() )という引数でイテレータ版を呼び出したのと等価です。つまり、コンテナとして一時オブジェクトへの参照を渡すことはできません。

```
iterator begin() const;
iterator end() const;
```

分割されたトークン列の、先頭と末尾を指すイテレータを返します。

## token_iterator

| 必要なヘッダファイル | #include <boost/tokenizer.hpp> |
|---|---|
| 名前空間 | boost |

```
template<typename TokenizerFunc = char_delimiters_separator<char>,
         typename Iterator      = std::string::const_iterator,
         typename Type          = std::string>
  class token_iterator_generator
  {
  public:
      typedef … type;
  };

template<typename Type, typename Iterator, typename TokenizerFunc>
```

```
typename token_iterator_generator<TokenizerFunc,Iterator,Type>::type
  make_token_iterator( Iterator begin, Iterator end,
                       const TokenizerFunc& f );
```

token_iterator_generatorは、トークン列を走査するイテレータ型を定義するための補助テンプレートです。必ずtoken_iterator_generator<…>::typeの形で使用して、イテレータ型を得ます。テンプレート引数のIterator型の種類がInputIterator(operator++と、operator*による読み取りができるイテレータ)であれば、定義されるイテレータもInputIteratorになります。Iterator型の種類がForwardIterator以上であれば、定義されるイテレータはForwardIteratorになります。

この型のイテレータを構築するには、make_token_iterator関数を使用します。また、トークン列の末尾を表すイテレータを選るには、token_iterator_generator<…>::typeをデフォルトコンストラクタで構築します。

## TokenizerFunctionコンセプト

以下の二つのpublicメンバ関数を持つ型は、tokenizerなどのテンプレートの第一引数として渡すことができます。言い方を変えれば、この二つの関数さえ実装すれば、ユーザ定義のtokenizerとして使うことが可能です。

```
class
{
public:
    void reset();
    template<typename Iterator, typename Token>
      bool operator()( Iterator& next, Iterator end, Token& tok );
};
```

operator()では、nextではじまりendを終端とする範囲からトークンを一つ切り出して、tokにその値を格納して下さい。nextはそのトークンの分だけ前進させます。トークンが取り出せなかった場合には、返値としてfalseを返します。

## 定義済み TokenizerFunction

| 必要なヘッダファイル | #include <boost/tokenizer.hpp> |
|---|---|
| 名前空間 | boost |

### char_separator

```
template< typename Char, typename Traits=std::char_traits<Char> >
  class char_separator
  {
  public:
    explicit char_separator();
    explicit char_separator(
                const Char*              dropped_delimiters,
                const Char*              kept_delimiters         = "",
                empty_token_policy empty_tokens = drop_empty_tokens );
  };
```

区切り文字を指定して文字列を分割するための TokenizerFunction です。第一引数と第二引数には区切りと見なす文字集合を、第三引数には空文字列をトークンとして扱うかどうかを、boost::drop_empty_tokens と boost::keep_empty_tokens のどちらかで指定します。第一引数と第二引数の違いは、後者で指定された区切り文字は、それ自体も有効なトークンとして扱われるという点です。

### escaped_list_separator

```
template< typename Char, typename Traits=std::char_traits<Char> >
  class escaped_list_separator
  {
  public:
    explicit escaped_list_separator(
                Char e = '¥¥', Char c = ',',Char q = '¥"' );
    escaped_list_separator(
                string_type e, string_type c, string_type q );
  };
```

エスケープ文字・区切り文字・引用符文字を指定して、コンマ区切りテキストをフィールド毎に分割するためのTokenizerFunctionです。複数の区切り文字を指定するには、二番目のコンストラクタで、複数文字を含んだ文字列として指定します。

## offset_separator

```
class offset_separator
{
public:
  template<typename Iter>
    offset_separator( Iter begin, Iter end,
                      bool loop_offsets    = true,
                      bool return_lastpart = true );
};
```

分割位置を表す整数のリストを指定して文字列を分割するためのTokenizerFunctionです。最初の二つの引数によって、分割位置を指定します。`loop_offsets`を`true`にすると、位置リストは繰り返すものとして扱われます。`return_lastpart`を`true`にすると、文字列の最後に余った部分を正常トークンとして認識するようになります。

## char_delimiters_separator

```
template< typename Char, typename Traits=std::char_traits<Char> >
  class char_delimiters_separator
  {
  public:
    explicit char_delimiters_separator(
                bool return_delims        = false,
                const kept_delims    = 0,
                const dropped_delims = 0 );
  };
```

非推奨です。代わりに`char_separator`を使うことが推奨されています。

`kept_delims`と`dropped_delims`で指定された文字が分割位置として使用されます。これら二つの引数は、省略するとそれぞれ`std::isspace`関数と`std::ispunct`関数が`true`となる文字が使用されます。

# 1.2 Xpressive(基本処理)

Boost C++ Libraries Programming 正規表現

標準ライブラリのstd::stringの提供する検索、置換処理は、基本的には「指定された文字列と完全にピッタリ一致する部分を探す」的なキッチリした検索です。もっと柔軟な条件による検索を実現するのが、「正規表現ライブラリ」Boost.Xpressiveです。

「正規表現」とは、文字列のパターンを記述する表記法です。例えば、

- 'a'で始まって'a'で終わるようなアルファベット列
- 行末の空白文字列

といったパターンは、正規表現では、

- a[a-z]*a
- ¥s+$

と表します。このようなパターン指定を受け取って、文字列から条件を満たす部分を検索といった強力な作業をおこなってくれるのが、正規表現ライブラリです。Boost.Xpressiveのサポートする正規表現については、49ページから始まるリファレンスをご覧ください。

では、早速使い方を見ていきます。最初の例として、ユーザーに何かの長さの値を入力させたいというシチュエーションを考えます。入力文字列がちゃんと長さをイメージできる形式で書かれているかどうかをチェックする処理を正規表現で書くと、次のようになります。

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

int main()
{
    // 数字が続いて、最後に"m"か"cm"か"mm"がつく文字列にマッチする正規表現
    //   "¥¥d" = 数字,    "+" = 1個以上並ぶ,    "|" = 選択肢のどれか
    cregex re = cregex::compile( "¥¥d+(m|cm|mm)" );
```

```
    // いろいろな文字列をテスト
    if( regex_match( "129cm",   re ) ) cout << "129cm - ok" << endl;
    if( regex_match( "42km",    re ) ) cout << "42km - ok" << endl;
    if( regex_match( "12345mm", re ) ) cout << "12345mm - ok" << endl;
    if( regex_match( "10c",     re ) ) cout << "10c - ok" << endl;

    // 途中までのテスト
    if( regex_match( "10c", re, regex_constants::match_partial ) )
              cout << "10c - partial" << endl;
}
```

▲Src. 1-6　xpressive1.cpp

実行結果

```
129cm - ok
12345mm - ok
10c - partial
```

ここでは、入力文字列としてm単位・cm単位・mm単位のいずれかのみを想定しました。想定する文字列パターンを`cregex::compile`関数で正規表現としてコンパイルし、`regex_match`関数で検査します。パターンと文字列が正しく合っている(マッチしている)ときは`true`が返され、マッチしていないときは`false`が返されています。

`regex_match`関数は、マッチの方針をフラグで細かく指定することが可能で、例えば`match_partial`フラグを指定すると、「完全にマッチはしていないけれど、あと何文字か付け加えればマッチする」場合にも`true`を返すような処理を記述できます。ユーザーが入力途中の文字列を随時チェックする処理などに応用が利くと思われます。

さて、次の例は、入力文字列全体とのマッチを取るのではなく、正規表現に合うような部分を検索する処理です。HTMLのタグのような、'<'で始まって'>'で終わる部分を文字列中から検索します。

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

```
using namespace boost::xpressive;

int main()
{
    // "<"で始まり">"以外の文字が続いて、">"で終わる文字列にマッチする正規表現
    //    "[^>]" = '>'以外の文字
    sregex re = sregex::compile( "<[^>]+>" );
    string str = "This is a <em>pen</em>.";

    // マッチする部分があるかどうか検索
    if( regex_search( str, re ) )
        cout << "found" << endl;

    // マッチする箇所を先頭から順に列挙
    string::const_iterator it=str.begin(), end=str.end();
    while( it != end )
    {
        smatch result;
        if( !regex_search(it, end, result, re) )
            break;
        cout << result.str() << endl;
        it = result[0].second;
          // result[0]はマッチ部位全体を表す情報
          // firstがマッチ位置の先頭、secondが末尾を指している
    }
}
```

▲Src. 1-7 xpressive2.cpp

実行結果

```
found
<em>
</em>
```

文字配列char*に対して正規表現処理をおこなう場合は、cregex::compile関数でcregex型の正規表現オブジェクトを作りました。std::string文字列に対して正規表現処理をおこなう場合は、sregex::compile関数で、sregex型の正規表現オブジェクトを作ります。(更にワイド文字を扱う場合は、wchar*に対応するwcregexクラス、std::wstringに対応するwsregexクラスを使うことになります。)正規表現そのものを表すオブジェクトはこのように種類が分かれていますが、正規表現処理を実行する関数(regex_matchやregex_searchなど)は、すべ

て同じ名前で使うことができます。

マッチした部分の情報は、cregexの場合はcmatchクラス、sregexの場合はsmatchクラスに格納されます。これらマッチ結果クラスには、マッチした位置やその部分文字列、"( )"を使った複雑な正規表現ならば部分マッチ情報などが格納されています。

なお、すべてのマッチ部分を順々に列挙するという処理は非常に頻繁に必要となりますから、regex_searchをラップした列挙用イテレータが提供されています。これを使うともっと簡単に全検索が実現できます。

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

int main()
{
    sregex re = sregex::compile("<[^>]+>");
    string str = "This is a <em>pen</em>.";

    // reにマッチする部分を順に列挙するイテレータ
    sregex_iterator it( str.begin(), str.end(), re );
    sregex_iterator end;
    for( ; it!=end; ++it )
        cout << it->position() << ": " << it->str() << endl;
}
```

▲Src. 1-8 xpressive3.cpp

実行結果

```
10:<em>
17:</em>
```

std::stringを全検索するイテレータはsregex_iterator型で、smatchを参照します。(const char*を全検索するイテレータはcregex_iterator型です。)

次は、正規表現とマッチする部分を検索した後、さらにそのマッチ箇所の置換をおこなう例です。

```
#include <iostream>
#include <string>
```

```
#include <boost/xpressive/xpressive.hpp>
using namespace std;
using namespace boost::xpressive;

int main()
{
    // 大文字のGで始まる単語を!!でくくって強調
    sregex re = sregex::compile("G¥¥w+");
    string str = "Good morning, Mr.George!";
    cout << regex_replace( str, re, string("!!$&!!") ) << endl;
}
```

▲Src. 1-9 xpressive4.cpp

実行結果

```
!!Good!! morning, Mr.!!George!!!
```

大文字のGではじまる単語を!!でくくって強調しています。regex_replaceでは、書式化文字列を指定すると、マッチ結果を使って新しい文字列を生成します。$&は、マッチした部分文字列全体を意味しています。$&のような書式化指定は、Perlなどと同じ記号を使って記述できます。処理対象がstd::stringの時は、書式化文字列もstd::stringで指定します。

最後に、日本語文字列を扱う上での注意点があります。const char*やstd::stringを受け取るバージョンの正規表現関数は残念ながらSHIFT_JISやEUC-JPなどのマルチバイト文字列に対応していません。const wchar_t*やstd::wstringを使って、ワイド文字列で正規表現処理をおこなう必要があります。次にワイド文字列を使った例を示します。

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

int main()
{
    wsregex re = wsregex::compile( L"第([1-9])号" );
    wstring str = L"第2号好評発売中！";
    wstring replaced = regex_replace( str, re, wstring(L"第$1巻") );
    if( replaced == L"第2巻好評発売中！" ) cout << "OK" << endl;
```

```
}
```

▲Src. 1-10　xpressive5.cpp

実行結果

```
OK
```

# 1.3 Xpressive(静的正規表現)

Boost C++ Libraries Programming 正規表現

1.2.1項では、文字列をcompile関数でコンパイルして、正規表現オブジェクトを作りました。このコンパイル方式では、プログラムの実行時にcompile関数が動いて正規表現を解釈するので、Boost.Xpressiveではこれを「動的正規表現」と呼んでいます。この方式と対照的なもう一つの正規表現オブジェクト作成方法が、「静的正規表現」です。

早速、前節の最初のサンプルを静的正規表現で書き直してみましょう。

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

int main()
{
    // 数字が続いて、最後に"m"か"cm"か"mm"がつく文字列にマッチする正規表現
    //   _d = 数字,    + = 1個以上並ぶ,    | = 選択肢のどれか
    cregex re = +_d >> (as_xpr("m")|"cm"|"mm");

    // いろいろな文字列をテスト
    if( regex_match( "129cm",   re ) ) cout << "129cm - ok" << endl;
    if( regex_match( "42km",    re ) ) cout << "42km - ok" << endl;
    if( regex_match( "12345mm", re ) ) cout << "12345mm - ok" << endl;
    if( regex_match( "10c",     re ) ) cout << "10c - ok" << endl;

    // 途中までのテスト
    if( regex_match( "10c", re, regex_constants::match_partial ) )
              cout << "10c - partial" << endl;
}
```

▲Src. 1-11 xpressive6.cpp

実行結果

```
129cm - ok
12345mm - ok
10c - partial
```

実際のところ、違いは正規表現オブジェクトの作り方だけなので、1.2.1項のサンプル(Src. 1-6　xpressive1.cpp)と違うのは1行だけです。つまり、

```
cregex re = cregex::compile( "¥¥d+(m|cm|mm)" );
```

このコンパイル部分が、

```
cregex re = +_d >> (as_xpr("m")|"cm"|"mm");
```

こう変わりました。regex_matchやregex_search、cregex_iteratorなどなどその他の部分の使い方は動的でも静的でもまったく違いありません。違いは正規表現オブジェクトの作り方のみです。動的正規表現は、文字列で正規表現パターンを記述しますが、静的正規表現では、C++の式としてパターンを記述します。

_dは、Boost.Xpressiveが標準で用意している正規表現オブジェクトで、数字1文字とマッチします。(cregex::compile("¥¥d")とほぼ同じ物です。)_dの左についている+は、_dの1回以上の繰り返しとマッチする正規表現オブジェクトを生成します。(文字列で正規表現を表すときは+を後ろに付けますが、C++の式として表現する都合上、+は前に置きます。)>>演算子は、二つのパターンで順番にマッチする正規表現オブジェクトを生成します。as_xprは渡された文字列と完全に一致する文字列のみとマッチする正規表現オブジェクトで、|演算子は普通の正規表現と同じ、「左右のどちらかとマッチする」の意味です。"cm"や"mm"は|演算子でくっつけている相手が既にXpressiveの部品であるため、ライブラリ側で自動的にas_xpr("cm")等に変換してくれます。

他のサンプルも静的正規表現で書いてみましょう。「"<"で始まり">"以外の文字が続いて、">"で終わる文字列にマッチする正規表現」はこうなります。

```
// sregex::compile( "<[^>]+>" );
sregex re = '<' >> +~as_xpr('>') >> '>';
```

「大文字のGで始まる単語」は次の通りです。

```
// sregex::compile("G¥¥w+");
sregex re = 'G' >> +_w;
```

「第(1から9までの全角数字)号」は以下のようになります。正規表現の一部を部分マッチとして取るには、(s1=...)という記法を使用します。

```
    // wsregex::compile( L"第([1-9])号" );
    wsregex re = L'第' >> (s1= range(L'1', L'9')) >> L'号';
```

動的正規表現を静的正規表現の中に混ぜて使うことも可能です。

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

int main()
{
    // abc か def にマッチする正規表現
    cregex reD = cregex::compile("abc|def");
    // それを ( ) でくくったもの
    cregex reS = "(" >> reD >> ")";

    cout << regex_match( "abc", reS )   << endl;
    cout << regex_match( "(def)", reS ) << endl;
}
```

▲Src. 1-12 xpressive7.cpp

実行結果

```
0
1
```

さて、わざわざこのような見慣れない記法で正規表現を記述する利点はなんでしょう？ 動的正規表現に対する静的正規表現の利点はいくつかあります。

- 正規表現の文法ミスが、コンパイル時に見つかる
- 動的版より、10～20％程度高速
- 再帰的なパターンなど、複雑なパターンが書ける
- 文字列以外のデータのパターン検索にも使える

一つめの利点は明らかだと思います。括弧の対応を間違った正規表現を書いてしまったときなど、静的正規表現はC++コンパイラが構文を解釈するので、実行する前のコンパイル時にエラーで教えてくれます。また、この構文解析処理がコンパイル時に回るため、二つめの利点として、実際の実行も多少効率的になります。（逆に言うと、静的版の方がコンパイルに時間がかかるという欠点でもあります。）

三つめの利点は、コードで実際に説明した方が早いでしょう。「左右の括弧のバランスが取れているか(括弧が対になっているか)判定するパターン」を、静的正規表現では以下のように表現します。このような再帰的なパターンはいわゆる典型的な「正規表現」では記述できない例です。(言語やライブラリによっては、「拡張正規表現」として対応していることもあります。)

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

int main()
{
    sregex parens;
    parens = *('(' >> by_ref(parens) >> ')');

    string line;
    while( cout<<"# ", getline(cin, line) )
    {
        if( regex_match( line, parens ) )
            cout<<"Balanced"<<endl;
        else
            cout<<"Not Balanced"<<endl;
    }
}
```

▲Src. 1-13 xpressive8.cpp

実行結果

```
# ()
Balanced
# (()())
Balanced
# (()((
Not Balanced
```

「括弧のバランスが取れている」という条件は、「開き括弧1個と閉じ括弧1個の間に、括弧のバランスが取れた文字列が挟まっている」と再帰的に表現できます。このように、いま作っている最中の正規表現パターンを自分自身の中で再帰的に使いたい場合は、by_ref関数でくるんで埋め込みます。

```
    parens = *('(' >> by_ref(parens) >> ')');
```

by_ref を使わないと、parens に前に設定されていた正規表現（この場合は、空の正規表現）を埋め込むという意味になってしまうので注意してください。また、次のように

```
    // 間違い
    sregex parens = *('(' >> by_ref(parens) >> ')');
```

初期化式の中で by_ref で再帰参照することはできません。未初期化の正規表現オブジェクトを by_ref 関数に渡すことになってしまうからです。Src. 1-13 のように、変数宣言と値の設定を別にすることで対応します。

四つ目の利点は、文字列でパターンを書くという制約を捨てたことで、文字列以外にも正規表現検索が使えるようになるという点です。これもコードでお見せしましょう。整数の配列に対して「一桁の数値が連続している部分をすべて取り出す」という検索を実行する例です。

```
#pragma warning ( disable : 4996 )
#include <iostream>
#include <string>
#include <boost/xpressive/xpressive.hpp>
#include <boost/xpressive/traits/null_regex_traits.hpp>
using namespace std;
using namespace boost::xpressive;

int main()
{
    null_regex_traits<int> nul;
    basic_regex<int*> onedigs = imbue(nul)( +range(1, 9) );

    int array[] = { 123, 4, 5, 67, -8, 9 };
    regex_iterator<int*> it( &array[0], &array[6], onedigs );
    regex_iterator<int*> end;
    for( ; it!=end; ++it )
        cout << "[" << it->position() << ","
                    << it->position()+it->length() << ")" << endl;
}
```

▲Src. 1-14 xpressive9.cpp

実行結果

```
[1,3)
[5,6)
```

int配列に対する正規表現オブジェクトは、basic_regex<int*>で、マッチ結果はmatch_result<int*>、イテレータはregex_iterator<int*>を使います。一桁の数値、つまり1から9の範囲にある数値の連続は、静的正規表現では+range(1, 9)と書くことができます。文字以外に対する正規表現に関しては、ここで一つ手間が必要で、正規表現全体をimbue(null_regex_traits<対象の型>())( ... )でくくっておかないと動作しません。imbueは大文字小文字の基準などを決めるロケールを設定するための機能ですが、文字以外に対して使うときにはそのような文字種に関する情報は一切使わないということを指定するのがnull_regex_traitsです。

Xpressiveの静的正規表現は、Part Ⅱ Chapter 7で紹介するBoost.Spiritにもよく似ています。どちらも、C++の演算子を使ってパターンを構築しますし、再帰的なパターンが書ける点も同じです。両者の差は、想定される用途の違いにあります。Xpressiveはパターンにマッチする部分を文字列中から検索したり、イテレータで列挙するなどの、柔軟な文字列検索機能としての用途を主眼に据えています。これに対してSpiritは、意味アクションや木構造指定などを使って、文字列の構造を解析してパターンの要素ごとに処理をおこなう、いわゆる構文解析を目的としています。Spiritの方がより一般的な枠組みではありますが、正規表現に特化したXpressiveの方が、より簡単に使えるように作られているということもできます。

# 1.4 Regex

Boost C++ Libraries Programming 正規表現

Boostには、Xpressiveの他にもう一つ、Regexという正規表現ライブラリが含まれています。Xpressiveの方が後発であり、Regexの提供する機能は、ほぼXpressiveの動的正規表現の部分に含まれるようになっています。通常はXpressiveのみを使うので十分でしょう。

ただし、細かいところで、Regexの方が優れている部分も残っています。そこでこの節では非常に簡単に、XpressiveとRegexの比較と、Regexの使い方を紹介します。

**Tbl. 1-1 XpressiveとRegexの比較**

| ポイント | Xpressive | Regex |
|---|---|---|
| 要ビルド | ○インクルードのみで使える | ビルドが必要 |
| コンパイル時間 | × | ○速い |
| 短い文字列に対する処理速度 | ○速い | × |
| 長い文字列に対する処理速度 | × | ○速い |
| 動的正規表現 | △あり | ○あり |
| 静的正規表現 | ○あり | なし |

両ライブラリの一つ目の相違点は、利用コストです。Xpressiveはヘッダファイルをインクルードするだけで使えるライブラリです。一方、Regexはあらかじめライブラリファイルをビルドする手間が必要です。この点は、逆にいうと、利用するたびに全体をビルドするXpressiveと比べて、あらかじめ中身をビルドしておけるRegexの方が、使う時にコンパイル時間がかからないということでもあります。

二つめの相違点は、アルゴリズムの内部実装の差です。一般的な傾向として、Xpressiveは短い文字列に対して検索をかけるのに向いており、Regexは長い文字列に対して検索をかけるのに向いているとされています。といっても、使う正規表現の内容などにも依存しますので、実際に正規表現がボトルネックになっている場合にベンチマークをとって比較しなおすのが良いでしょう。

三つめの相違点は、動的正規表現でサポートされる表現の種類です。Xpressiveで使える表現はすべてRegexでもサポートされており、さらに、Regexでのみ使える表現があります。

**Tbl. 1-2　Regexのみで使える正規表現**

| | | | |
|---|---|---|---|
| ¥l | [:lower:]と同義。小文字 | ¥u | [:upper:]と同義。大文字 |
| ¥< | 単語先頭 | ¥> | 単語末尾 |
| ¥‘ | バッファ先頭 | ¥A | バッファ先頭 |
| ¥’ | バッファ末尾 | ¥z | バッファ末尾 |
| ¥Z | バッファ末尾またはバッファ末尾の改行 | | |

最後に、静的正規表現はXpressiveにしかありません。

Regexの関数やクラスの名前、使い方はXpressiveとほぼ同じです。（正確には、後発のXpressiveが、Regexのインターフェイスに合わせて設計されています。）最大にしてほぼ唯一の違いは、正規表現オブジェクトの作り方です。Xpressiveでは、マッチする対象ごとに、cregex、sregexなど異なる型が用意されていましたが、Regexではregexクラス1種類です。また、正規表現オブジェクトはcompile関数を呼び出して作るのではなく、コンストラクタにパターン文字列を渡して作成します。

```
// Regex の
//  sregex re = sregex::compile( "<[^>]+>" );
// を Xpressiveで書くと
regex re( "<[^>]+>" );
```

あとは、regex_matchやregex_searchをはじめとして、すべてXpressiveと同じ使い方ができます。regexは1種類にまとまりましたが、マッチの結果を表すcmatch、smatchやイテレータ型は別々なところも、Xpressiveと同様です。「'<'で始まって'>'で終わる部分を文字列中から検索する」サンプルを、Regexで書き直すと以下のようになります。

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

```
using namespace std;
using namespace boost;      // 名前空間は boost

int main()
{
    regex re( "<[^>]+>" );  // 正規表現オブジェクトはregex 1種類
                            // compile関数ではなく、コンストラクタで生成

    // あとはXpressiveとまったく同じ
    string str = "This is a <em>pen</em>.";
    if( regex_search( str, re ) )
        cout << "found" << endl;

    string::const_iterator it=str.begin(), end=str.end();
    while( it != end )
    {
        smatch result;
        if( !regex_search(it, end, result, re) )
            break;
        cout << result.str() << endl;
        it = result[0].second;
    }
}
```

▲Src. 1-15 regex1.cpp

# Reference

Boost C++ Libraries Programming 正規表現

## 正規表現の構文

Tbl. 1-3 文字

| 動的 | 静的 | 意味 |
|---|---|---|
| . | _ | 任意の一文字 |
| [aiueo] | (set='a','i','u','e','o') | a, i, u, e, o のどれか一文字 |
| [^aiueo] | ~(set='a','i','u','e','o') | a, i, u, e, o 以外の任意一文字 |
| [a-z] | range('a','z') | a から z の範囲の一文字 |
| [^a-z] | ~range('a','z') | a から z の範囲外の一文字 |
| [:classname:] | classname | 指定文字種の一文字。classname には、alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit のいずれかを指定する |
| ¥w | _w | 単語構成文字 |
| ¥W | ~_w | 単語構成文字以外 |
| ¥s | _s | [:space:]と同義。空白文字 |
| ¥S | ~_s | 空白文字以外 |
| ¥d | | [:digit:]と同義。数字 |
| ¥D | ~_d | 数字以外 |
| ¥n | _n | 改行文字 |
| ¥r?¥n | _ln | 論理的改行文字(¥r, ¥r¥n, ¥n のいずれもマッチする) |
| (?i:expr) | icase(expr) | 括弧の中では大文字小文字の違いを無視 |

Tbl. 1-4 繰り返し・分岐

| 動的 | 静的 | 意味 |
|---|---|---|
| a* | *a | aの0個以上の繰り返し |
| a+ | +a | aの1個以上の繰り返し |
| a? | !a | aの0個または1個の出現 |
| a{M,N} | repeat<M,N>(a) | aのM個以上N個以下の繰り返し |
| a{M} | repeat<M>(a) | aのM個の繰り返し |
| a\|b | a\|b | aまたはb |

繰り返しは通常できるだけ長いマッチが選択されますが、動的正規表現の場合繰り返し記号の後ろに'?'、静的正規表現の場合前に'-'をつけると、可能なマッチのうち最短のものが選択されます。(例："a*?", -*a)

Tbl. 1-5 部分式

| 動的 | 静的 | 意味 |
|---|---|---|
| (expr) | (s1=expr) | 部分式のグループ化 |
| (?:expr) | (expr) | 後方参照等が可能な部分式ではない、単なるグループ化 |
| ¥1,…,¥9 | s1,…,s9 | 後方参照 |

Tbl. 1-6 位置指定

| 動的 | 静的 | 意味 | 動的 | 静的 | 意味 |
|---|---|---|---|---|---|
| ^ | bos | 行頭 | $ | eos | 行末 |
| ¥b | _b | 単語頭または単語末 | ¥B | ~_b | 単語中 |

Tbl. 1-7 先読み

| 動的 | 静的 | 意味 |
|---|---|---|
| (?=expr) | before(expr) | 続く文字列が正規表現exprにマッチすれば、現在位置の0文字にマッチ |
| (?!expr) | ~before(expr) | 続く文字列が正規表現exprにマッチしなければ、現在位置の0文字にマッチ |
| (?>expr) | keep(expr) | バックトラック対象にならない独立マッチ |
| (?<=expr(?<=) | after(expr) | 直前の文字列が正規表現exprにマッチすれば、現在位置の0文字にマッチ |
| (?<!expr) | ~after(expr) | 直前の文字列が正規表現exprにマッチしなければ、現在位置の0文字にマッチ |

動的正規表現のための特殊な記号.|*?+(){}[]^$¥-は、¥でエスケープして利用します。(例："¥[¥d+¥]")

C++のソースコード中に文字列定数で正規表現を指定する際には、エスケープ文字'¥'に関して注意が必要です。文字列定数で文字'¥'を表すには'¥¥'と書く必要があるため、例えば単語を表す正規表現を指定するには、"¥w"ではなく"¥¥w"と記述する必要があります。

## 置換書式文字列の構文

Boost.Xpressiveでは、文字列と正規表現とのマッチ結果を整形して、文字列として出力することができます。整形方法は「置換書式文字列」と呼ばれる文字列をregex_replace関数やmatch_results::format関数に渡して指定します。

置換書式文字列中の以下で示す文字の並びは、整形の際に、マッチ結果情報を使って次のように置き換えられます。

**Tbl. 1-8　マッチ結果を用いた置換**

| | |
|---|---|
| $‘ | 直前のマッチ部分の終端(または入力文字列の先頭)から現在のマッチ位置の先頭までの文字列 |
| $’ | 現在のマッチ部分の終端から入力文字列の終端までの文字列 |
| $& | マッチした部分全体 |
| $N | N番目の部分マッチ。Nには1以上の整数を指定します。 |
| $$ | 文字$そのもの |

## mark_tag

| | |
|---|---|
| 必要なヘッダファイル | #include <boost/xpressive/xpressive.hpp> |
| 名前空間 | boost::xpressive |

```
struct mark_tag
{
    mark_tag(int n);
};
```

静的正規表現では、部分式に名前をつけるときには(s1=...)や(s2=...)の形を使用します。このs1やs2の代わりに、独自の名前を付けることを可能にするのがmark_tagです。整数Nで初期化したmark_tag型変数は、sNの代わりに使うことができます。

```
// 例：カンマの両側に同じ個数の'a'が並んでいるパターン
mark_tag aseq(1);
sregex re = "(" << (aseq= +as_xpr('a')) << "," << aseq << ")";
```

## basic_regex<BidiIter>クラステンプレート

| 必要なヘッダファイル | #include <boost/xpressive/xpressive.hpp> |
|---|---|
| 名前空間 | boost::xpressive |

```
template<typename BidiIter>
struct basic_regex
{
    basic_regex();
    basic_regex(const basic_regex<BidiIter>&);
    template<typename Xpr> basic_regex(const Xpr&);

    static basic_regex<BidiIter>
      compile(const string_type& regex,
              flag_type              f = regex_constants::ECMAScript) ;
};
```

正規表現のコンパイル結果を格納した構造体です。compile関数によって文字列から生成するか、静的正規表現を使って生成します。このクラスのオブジェクトをアルゴリズム関数に渡して、検索や置換などの文字列処理を実行します。

## 型定義

```
typedef basic_regex<const char*>                    cregex;
typedef basic_regex<std::string::const_iterator>    sregex;
typedef basic_regex<const wchar_t*>                wcregex;
typedef basic_regex<std::wstring::const_iterator> wsregex;
```

よく使われる正規表現クラスについて、短縮名が定義されています。

## コンパイル

```
static basic_regex< BidiIter >
  compile( const string_type& regex,
           flag_type              f = regex_constants::ECMAScript);
```

文字列で指定した正規表現をコンパイルして返します。不正な文字列を指定した場合、regex_error例外が送出されます。第2引数に指定するフラグは、以下の値の組み合わせを用います。

Tbl. 1-9　フラグ

| 名前 | 意味 |
|---|---|
| regex_constants::ECMAScript | ECMAScript拡張互換 |
| regex_constants::nosubs | 大文字小文字の違いを無視 |
| regex_constants::nosubs | (…)による部分式マッチを格納しない |
| regex_constants::optimize | 速度重視で正規表現をコンパイル(現在のBoost.Regexでは無意味) |
| regex_constants::collate | [a-z]などの範囲指定にロケールを反映 |
| regex_constants::single_line | ^と$が文字列内の改行文字とはマッチしないようにする |
| regex_constants::not_dot_null | .が¥0とマッチしないようにする |
| regex_constants::not_dot_newline | .が改行文字とマッチしないようにする |
| regex_constants::ignore_white_space | 正規表現内の空白文字を無視 |

## 検索・置換アルゴリズム関数

| 必要なヘッダファイル | #include <boost/xpressive/xpressive.hpp> |
|---|---|
| 名前空間 | boost::xpressive |

### マッチ方式を指定するフラグ

正規表現と文字列のマッチ処理の方式は、以下のフラグを組み合わせて指定します。

| | |
|---|---|
| regex_constants::match_default | ECMAScript言語使用で定義された規則に従うマッチ方式 |
| regex_constants::match_not_bol | "^"を文字列先頭の空文字列にはマッチさせない(not-beginnig-of-line) |
| regex_constants::match_not_eol | "$"を文字列末尾の空文字列にはマッチさせない(not-end-of-line) |
| regex_constants::match_not_bow | "¥b"を文字列先頭の空文字列にはマッチさせない(not-begining-of-word) |
| regex_constants::match_not_eow | "¥b"を文字列末尾の空文字列にはマッチさせない(not-end-of-word) |
| regex_constants::match_any | 複数のマッチがある場合、どれも結果として許可する |
| regex_constants::match_not_null | 正規表現は長さ0の部分文字列にはマッチさせない |
| regex_constants::match_continuous | マッチは、入力文字列の先頭から始まる部分に限る |
| regex_constants::match_partial | 後ろに適切な文字列を付け加えればマッチできるような部分列(空文字列は除く)もマッチとして許可する |
| regex_constants::match_prev_avail | 入力文字列の1文字前へのアクセス(*--first)を許し、"^"や"¥b"などを直前の文字情報に基づき判定 |
| regex_constants::format_default | 置換はECMAScript言語仕様で定義された動作で実行。すなわち、入力文字列中の重なりのないマッチ箇所はすべて書式文字列に従って置換し、マッチしなかった箇所はそのままコピーして出力 |

| | |
|---|---|
| regex_constants::format_no_copy | 正規表現にマッチしなかった部分は、置換結果に出力しない |
| regex_constants::format_first_only | 最初に見つかったマッチ部分のみを置換 |
| regex_constants::format_literal | 書式指定に従って置換するのではなく書式文字列そのもので置換する |

フラグ定数の型はregex_constants::match_flag_typeです。

## regex_match

```
template<class BidiIter>
  bool regex_match( 文字列,
    const basic_regex<BidiIter>& re,
    match_flag_type               f = regex_constants::match_default );

template<class BidiIter>
  bool regex_match( 文字列,
    match_results<BidiIter>&      m,
    const basic_regex<BidiIter>& re,
    match_flag_type               f = regex_constants::match_default );
```

入力文字列全体と正規表現がマッチするかどうかを検査します。マッチすればtrue、マッチしなければfalseを返します。詳細なマッチ結果が必要な場合は、第2引数としてmatch_results型変数への参照を渡します。*文字列*の部分は、以下のいずれかの形式で指定します。

- ¥0終端文字列へのポインタ
- std::basic_string型の文字列
- 文字列先頭と終端を指す二つのイテレータ

## regex_search

```
template<class BidiIter>
  bool regex_search( 文字列,
    const basic_regex<BidiIter>& re,
    match_flag_type               f = regex_constants::match_default );

template<class BidiIter>
  bool regex_search( 文字列,
```

```
    match_results<BidiIter>&      m,
    const basic_regex<BidiIter>& re,
    match_flag_type               f = regex_constants::match_default );
```

入力文字列の中から、正規表現がマッチする最初の部分を検索します。見つかればtrue、見つからなければfalseを返します。詳細なマッチ結果が必要な場合は、第2引数としてmatch_results型変数への参照を渡します。*文字列*の部分は、regex_match関数と同様の形式で指定します。

## regex_replace

```
template<class Char>
  std::basic_string<Char> regex_replace( 文字列,
    const basic_regex<BidiIter>&        re,
    const std::basic_string<Char>&  format,
    match_flag_type f = regex_constants::match_default );

template<class OutIter, class BidiIter>
  OutIter regex_replace( OutIter out,
    文字列,
    const basic_regex<BidiIter>&        re,
    const std::basic_string<Char>&  format,
    match_flag_type f = regex_constants::match_default );
```

入力文字列中の正規表現にマッチした部分を、置換書式文字列formatをもとに置換します。一つめの形式では、*文字列*としてはstd::basic_string型の文字列のみを渡すことができ、返値も同じ型で返ってきます。二つめの形式では、*文字列*には二つのイテレータを指定でき、結果は第1引数に指定したイテレータに書き込まれます。

## match_results<BidiIter> クラステンプレート

| 必要なヘッダファイル | #include <boost/xpressive/xpressive.hpp> |
|---|---|
| 名前空間 | boost::xpressive |

```
template<typename BidiIter>
struct match_results
{
      bool      empty()    const;
      size_type size()     const;

      difference_type   length( int sub = 0 ) const;
      difference_type position( int sub = 0 ) const;
      string_type          str( int sub = 0 ) const;

      const_iterator begin() const;
      const_iterator end() const;
      const_reference operator[]( int sub ) const;

      const nested_results_type& nested_results() const;

      const_reference prefix() const;
      const_reference suffix() const;

      template<class OutIter>
        OutIter format( OutIter            out,
                        const string_type& format,
                        match_flag_type    f = format_default ) const;
      string_type format( const string_type& format,
                          match_flag_type    f = format_default ) const;
};
```

正規表現と文字列とのマッチ結果を返すためのデータ構造です。マッチした部分に関する情報や、正規表現中の括弧でくくられた部分式にマッチした部分の情報を取得できます。sub=0 とすると正規表現全体についての情報、それ以外の場合は sub 番目の部分式についての情報が返却されます。sub 引数には、mark_tag を指定することもできます。

## 型定義

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

よく使われる正規表現結果クラスについて、短縮名が定義されています。

## 部分式アクセス

```
const_iterator  begin() const;
const_iterator  end() const;
const_reference operator[]( int sub ) const;

const nested_results_type& nested_results() const;
```

begin, endの返すイテレータやoperator[]の参照先はsub_match<BidiIter>クラスのオブジェクトです。

また、by_refで複数の正規表現オブジェクトを組み合わせた場合、nested_resultsによって内側の正規表現の部分マッチを取得することができます。nested_results().begin()からnested_results.end()まで、イテレータで内側のmatch_results構造体を取得することができます。

## マッチの前後へのアクセス

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

prefixは、元の文字列の先頭から、最初のマッチ位置までの部分文字列を指すsub_matchオブジェクトを返します。suffixは、最後のマッチから元の文字列終端までを返します。

## sub_match<BidiIter>クラステンプレート

| 必要なヘッダファイル | #include <boost/xpressive/xpressive.hpp> |
|---|---|
| 名前空間 | boost::xpressive |

```
template<typename BidiIter>
  struct sub_match
      : public std::pair<BidiIter, BidiIter>
  {
       string_type str() const;
       operator string_type() const;
       difference_type length() const;

       operator bool_type() const;
       bool operator!() const;
       bool matched;
  };

// 他に、比較演算子・ストリーム出力演算子が定義されています。
```

正規表現と文字列の一つ一つのマッチ部分を表すクラスです。

## regex_iterator<BidiIter>クラステンプレート

| 必要なヘッダファイル | #include <boost/xpressive/xpressive.hpp> |
|---|---|
| 名前空間 | boost::xpressive |

```
template<typename BidiIter>
  struct regex_iterator
  {
      regex_iterator();
      regex_iterator(BidiIter, BidiIter,
        const basic_regex<BidiIter>& re,
        match_flag_type f = regex_constants::match_default);

      typedef match_results<BidiIter>  value_type;
      const value_type& operator *() const;
      const value_type* operator->() const;
```

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

正規表現と文字列のマッチとして可能な部分をすべてmatch_result型で列挙するためのイテレータです。ForwardIteratorとしての条件を満たしています。コンストラクタにはイテレータを二つ渡して文字列を指定し、reで正規表現オブジェクト、mでフラグを指定します。終端マーカにあたるイテレータはデフォルトコンストラクタで作成します。

### 型定義

```
typedef regex_iterator<const char*>                    cregex_iterator;
typedef regex_iterator<std::string::const_iterator>    sregex_iterator;
typedef regex_iterator<const wchar_t*>                 wcregex_iterator;
typedef regex_iterator<std::wstring::const_iterator> wsregex_iterator;
```

よく使われる正規表現イテレータクラスについて、短縮名が定義されています。

## regex_token_iterator<BidiIter>クラステンプレート

| 必要なヘッダファイル | #include <boost/xpressive/xpressive.hpp> |
|---|---|
| 名前空間 | boost::xpressive |

```
template<typename BidiIter>
  struct regex_iterator
  {
      regex_token_iterator();
      template<typename SubMatches>
      regex_token_iterator(BidiIter, BidiIter,
        const basic_regex<BidiIter>& re,
        const SubMatches&            sub,
        match_flag_type f = regex_constants::match_default);

      typedef iterator_value< BidiIter >::type      char_type;
      typedef std::basic_string< char_type >        value_type;

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

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

正規表現と文字列のマッチとして可能な部分をすべて列挙し、その結果を文字列で列挙するためのイテレータです。ForwardIteratorとしての条件を満たしています。コンストラクタにはイテレータを二つ渡して文字列を指定し、reで正規表現オブジェクト、mでフラグを指定します。終端マーカにあたるイテレータはデフォルトコンストラクタで作成します。

引数subには、マッチ結果のどの部分を文字列として列挙するかを指定します。

| 動的 | 静的 |
|---|---|
| 0 | 正規表現にマッチした部分を列挙 |
| -1 | 正規表現にマッチしなかった部分を列挙 |
| i(1以上のint) | 正規表現内のi番目のグループにマッチした部分を列挙 |
| {i,j,k}(int配列) | 正規表現内のi番目→j番目→k番目のグループにマッチした部分を列挙 |

指定-1は、正規表現で区切り文字を表現して、そのパターンで区切られた項目を取り出すといった用途に使うことができます。

## 型定義

```
typedef regex_iterator<const char*>
  cregex_iterator;
typedef regex_iterator<std::string::const_iterator>
  sregex_iterator;
typedef regex_iterator<const wchar_t*>
  wcregex_iterator;
typedef regex_iterator<std::wstring::const_iterator>
  wsregex_iterator;
```

よく使われる正規表現トークンイテレータクラスについて、短縮名が定義されています。

# 1.5 lexical_cast

Boost C++ Libraries Programming　データの文字列変換

整数型の値を文字列に変換したり、逆に、数字が並んだ文字列を整数型の値に変換したりしたいという場合は結構よくあると思います。例えば

- 計算結果を画面上のテキストボックスに表示したい
- テキスト形式の設定ファイルに数値データを読み書きしたい

もちろん整数型だけでなく、浮動小数点数型や、自分で作ったクラスのオブジェクトなど、どんなデータも、文字列として扱えると非常に便利です。

標準C++ライブラリを使ってこれらの変換を実現するには、大きく分けて3つの方法がありました。

```
const char* in_str = "65536";
int number = atoi( in_str );
```

▲Src. 1-16　atoi, atof系関数を使う方法

```
const char* in_str = "3.14";

float x;
sscanf( in_str, "%f", &x );

char out_str[20] = "";
snprintf( out_str, sizeof(out_str)-1, "%f", x*x );
```

▲Src. 1-17　sprintf, sscanf系関数を使う方法

```
int number = -12345;
stringstream ss;
ss << number;
string out_str = ss.str();
```

▲Src. 1-18　stringstreamクラスを使う方法

しかし、それぞれ欠点があります。atoiやatofは簡単で使いやすいですが、文字列から整数か浮動小数点数を取り出すときにしか使用できません。ユーザ定義のクラスで同じことを実現するには、atoXxxやatoYyyのようなクラス毎に違う名前の関数を定義して、使い分けなくてはなりません。それにそもそも、これらには

値から文字列への変換機能が無いです。sprintf や sscanf による方法は、強力な書式指定という魅力がありますが、型に関する安全性に欠けます。また、一時変数が必要なので、atoi のように他の式の中に埋め込んで手軽に使うことができません。

stringstream クラスを使う方法には、上の二つと比べて良い点があります。それは、どんな型でも、operator>> と operator<< さえ定義してあれば stringstream クラスで扱えるということ。型によらない汎用のテンプレートを書いているときなど、この特徴は重宝します。stringstream クラスの問題点は、これもやはり一時変数が必要なこと。コード中で文字列変換が必要になるたびに本筋の処理と関係ない stringstream ss;という変数が現れたら、ちょっとと読みにくいです。

Boost の lexical_cast は、stringstream クラスのような拡張性をもち、しかも無駄な変数の必要ない文字列変換法を提供します。

```
string in_str = "3.14";
float  f       = lexical_cast<float>( in_str );

int    number = -12345;
string out_str = lexical_cast<string>( number );
```

▲Src. 1-19　Boost の lexical_cast を使う方法

lexical_cast<型名>( 文字列 )で、文字列から指定の型への変換になります。逆に、lexical_cast<std::string>( 式 )で、式の結果を文字列に変換することができます。

他の例も見てみましょう。

```
#include <iostream>
#include <boost/lexical_cast.hpp>
using namespace std;
using namespace boost;

int main( int argc, char* argv[] )
{
    if( argc <= 1 )
        return -1;

    try
    {
        int N = lexical_cast<int>( argv[1] );
        for(int i=0; i!=N; ++i)
```

```
            cout << "Hello, World." << endl;
    }
    catch( const bad_lexical_cast& e )
    {
        cerr << "整数を入れてください"   << endl
             << "-- " << e.what()     << endl;
        return -1;
    }

    return 0;
}
```

▲Src. 1-20 lexcast1.cpp

実行結果

```
> lexcast1 3
Hello, World.
Hello, World.
Hello, World.
> lexcast1 NonNumberString
整数を入れてください
-- bad lexical cast: source type value could not be interpreted as target
```

コマンド引数として渡された文字列を数値に変換して、その回数だけ"Hello, World."を画面に表示しています。lexical_castでは、文字列の内容が変換先の型に合っていないとき、bad_lexical_cast例外が発生します。変換に失敗する恐れがあるときは、この例外をcatchして適宜処理するコードを書くことになるでしょう。

stringstreamクラスを使った場合と同様に、operator>>とoperator<<を定義すれば、どの型でもlexical_castが使えるようになります。

```
#include <iostream>
#include <fstream>
#include <string>
#include <boost/lexical_cast.hpp>
using namespace std;
using namespace boost;

// 日付
struct Date
{
```

```
    int month;
    int date;
    Date( int m, int d ) : month(m), date(d) {}
};

// 自分の作ったDate構造体用に、<<演算子を定義
ostream& operator<<( ostream& os, const Date& d )
{
    return os << d.month << '-' << d.date;
}

int main()
{
    Date today(12,4);

    // "diary-12-4"のようなファイル名文字列を作るために、
    // lexical_castを使って日付を文字列に変換しています。
      string filename = "diary-" + lexical_cast<string>( today );
    ofstream fout( filename.c_str() );

    fout << "今日は楽しい一日でした。" << endl;
    return 0;
}
```

▲Src. 1-21　lexcast2.cpp

またlexical_castでは、型によらず同じ形の呼び出しで文字列へ変換ができるので、文字列化できる型なら何でも使えるテンプレートを書くのも簡単です。

```
class GenericListBox
{
    // 文字列のリストを画面に表示して、マウスやキーボードで
    // 選択可能にするGUI部品クラスが提供されているとします。
    GUI::ListBox impl;
    ...

public:
    // その部品をラップして、文字列化できる任意の型を
    // リストボックスで提示できるようにします。
    void addItem( const T& item )
    {
        impl.addItem( lexical_cast<string>(item) );
    }
```

```
    bool isSelected( const T& item ) const
    {
        impl.isSelected( lexical_cast<string>(item) );
    }
    ...
};
```

▲Src. 1-22 特定の型を気にせずlexical_cast

ところで余談になりますが、お気づきでしょうか？ このlexical_castの書き方は、C++組み込みのキャスト演算子ととてもよく似た形をしています。

```
Derived* pDerived   = dynamic_cast<Derived*>( pBase );
unsigned int lparam = reinterpret_cast<unsigned int>( pDerived );
string s            = lexical_cast<string>( lparam );
```

▲Src. 1-23 C++のキャストとlexical_cast

どれが言語で規定された演算子でどれがライブラリ関数なのか、もはや見た目では全く区別がつきません。言い方を変えると、ユーザー定義の変換関数であるlexical_castが、キャストとして実にうまくC++言語に溶け込んでいます。lexical_castを知ったとき私は、なるほどこういう理由があってdynamic_castなどはあんな形で定義されていたのか！ と感心したものでした。

# Reference



## lexical_cast関数

| 必要なヘッダファイル | #include <boost/lexical_cast.hpp> |
|---|---|
| 名前空間 | boost |

```
template<typename Target, typename Source>
  Target lexical_cast( Source arg )

class bad_lexical_cast : public std::bad_cast;
```

引数argを、文字列表現を仲立ちとしてTarget型へ変換します。通常はSourceの型はコンパイラに推論させて、lexical_cast<Target>( arg )の形で呼び出します。operator<<( ostream&, Source )とoperator>>( istream&, Target )の二つの演算子が定義されている必要があります。また、Source型とTarget型双方にコピーコンストラクタ、Target型にはデフォルトコンストラクタが必要です。

argの文字列表現とTarget型の要求する文字列表現が一致しなかった場合には、bad_lexical_cast例外が送出されます。

# 1.6 format

Boost C++ Libraries Programming　データの文字列変換

一言で説明してしまうと、C++風にアレンジされたprintfです。まずはサンプルプログラムをご覧下さい。

```
#include <iostream>
#include <boost/format.hpp>
using namespace std;
using namespace boost;

int main()
{
    double x=1.23, y=4.56;
    cout << format("(%1%, %2%)") % x % y << endl;
}
```

▲Src. 1-24　format1.cpp

実行結果

```
> format1
(1.23, 4.56)
```

文字列"(%1%, %2%)"は「書式指定文字列」と呼ばれるもので、出力のフォーマットを指定しています。この例の%1%は「ここに一個目の引数を書く」、%2%は「ここに二個目の引数を書く」という意味です。%で始まる命令の部分以外は、文字がそのまま出力されます。書式指定文字列につづけて%記号で区切って引数を渡すと、指定に従って整形された文字列が得られます。

整理しましょう。formatライブラリは、次の形で使います。

```
format(書式指定文字列) % 引数1 % 引数2 % ... % 引数N
```

▲Src. 1-25　Boost.Formatの使い方の一般形

できあがった文字列を標準出力に書き出したいなら、もちろん次のように書きます。

```
cout << format(書式指定文字列) % 引数1 % 引数2 % ... % 引数N;
```

結果をいったん変数に格納するときは、boost::io::str関数で文字列化します。

```
using boost::io::str;
string s = str( format(書式指定文字列) % 引数1 % 引数2 % ... % 引数N );
```

この例をご覧になって、もしかしたら「% 引数 % 引数 ...」にビックリされた方もいらっしゃるのではないでしょうか。このライブラリでは、割り算の余りの計算に使われる%演算子を多重定義して、formatへの引数渡しという役割を担わせています。この一見不思議な記法は、ただ「書式指定の'%'と同じ記号でキュートでしょ？」というだけではありません。実はちゃんとした理由を持っています。

それは、可変個の引数を安全に受け取るため、です。C++では三連ドット記号(...)を使って可変個の引数を受け取る関数が定義できますが、これを使って引数を受け渡しすると、引数の個数と型の情報が完全に失われてしまいます。標準ライブラリのprintf関数では書式指定文字列を解析してその情報をとりもどしていますが、これは、もしもプログラマが間違った書式指定をしてしまったら、大変なことになります。printf関数は、本当はただの整数なのに文字列へのポインタとしてデータを扱ったり、本当は2個しか引数を渡していないのに3個目に触ろうとしてしまったり、もう何が起きるかわからないトワイライトゾーンへ突入してしまいます。%演算子で1個ずつ引数を渡していく方式なら、この心配はありません。

さて、では次に、書式指定の方法について見ていきましょう。まず、formatの書式はprintfの上位互換になっているので、printfで使えた指定は全部認識されます。

```
#include <iostream>
#include <boost/format.hpp>
using namespace std;
using namespace boost;

int main()
{
    int iX, iY;
    cout << "何か数を入力してください： "; cin >> iX;
    cout << "もいっちょ： ";               cin >> iY;

    cout << format("%d + %d = %d") % iX % iY % (iX+iY) << endl;
    cout << format("%d - %d = %d") % iX % iY % (iX-iY) << endl;

    cout << "8進数で書くと" << endl
         << format("%o + %o = %o") % iX % iY % (iX+iY) << endl;
```

```
    cout << "16進数で書くと" << endl
         << format("%02X + %02X = %02X") % iX % iY % (iX+iY) << endl;

    cout << "16進数で書くとその2" << endl
         << format("%#02x + %#02x = %#02x") % iX % iY % (iX+iY) << endl;

    double divided = double(iX) / double(iY);
    cout << "割り算の結果を4桁まで表示" << endl
         << format("%d / %d = %+.4f") % iX % iY % divided << endl;
}
```

▲Src. 1-26 format2.cpp

実行結果

```
> format2
何か数を入力してください： 12
もいっちょ： 111
12 + 111 = 123
12 - 111 = -99
8進数で書くと
14 + 157 = 173
16進数で書くと
0C + 6F = 7B
16進数で書くとその2
0xc + 0x6f = 0x7b
割り算の結果を4桁まで表示
12 / 111 = +0.1081
```

それぞれの指定の意味はP.77のリファレンスをどうぞ。

さらにprintfの拡張として、何番目の引数を展開するのかを指定できるようになっています。これを利用すると、引数と逆の順番で文字列を並べたり、一個の引数を複数回別の書式で表示したりすることができます。

```
#include <iostream>
#include <boost/format.hpp>
using namespace std;
using namespace boost;

int main()
{
    // 順番入れ替え
```

```
    cout << format("%3% %2% %1%") % "いち" % "にい" % "さん" << endl;
    // 途中の引数を使わない
    cout << format("%1% %3%") % "いち" % "にい" % "さん" << endl;
    // 2回同じ引数を展開
    cout << format("%1% %3% %1% %2%") % "いち" % "にい" % "さん" << endl;

    // 桁数など出力の細かい指定は、引数番号のうしろに
    // '$'を書き、それに続けてprintfと同様の書式で記述する。
    cout << format("16進=%1$x  10進=%1$d  8進=%1$o") % 15 << endl;
    cout << format("%.7e") % (atan(1.0)*4) << endl;
}
```

▲Src. 1-27　format3.cpp

実行結果

```
> format3
さん にい いち
いち さん
いち さん いち にい
16進=f  10進=15  8進=17
3.1415927e+000
```

この順番を変えられるという性質は、例えば多言語対応のエラーメッセージを用意する、なんて状況を考えると非常に役に立ちます。

```
#include <iostream>
#include <cmath>
#include <boost/format.hpp>
using namespace std;
using namespace boost;

namespace err
{
    enum {Japanese, English};
    enum {NO_FILE, SECTION_BROKEN, /*...*/};

    static const char* message[][2] =
    {
        { "ファイル '%1%' を開けませんでした。",
          "File '%1%' could not be opened."
        },
        { "ファイル '%1%' 内のセクション [%2%] が破損しています。",
          "Section [%2%] in file '%1%' is broken."
```

```
        },
        // ...
    };
}

int main(int argc, char* argv[])
{
    int lang = err::Japanese;
    if( argc==2 && argv[1][0]=='e' )
        lang = err::English;

    // ...

    format fmtr( err::message[err::SECTION_BROKEN][lang] );
    cerr << fmtr % "data.archive" % "top_secret" << endl;
}
```

実行結果

```
> format4
ファイル 'data.archive' 内のセクション [top_secret] が破損しています。
> format4 e
Section [top_secret] in file 'data.archive' is broken.
```

# Reference



## basic_format<charT,Traits>クラステンプレート

| 必要なヘッダファイル | #include <boost/format.hpp> |
|---|---|
| 名前空間 | boost |

```
template< typename CharT, typename Traits=std::char_traits<CharT> >
  class basic_format
  {
  public:
      typedef std::basic_string<CharT, Traits> string_t;
      basic_format( const CharT* fmt );
      basic_format( const CharT* fmt, const std::locale& loc );
      basic_format( const string_t& fmt );
      basic_format( const string_t& fmt, const std::locale& loc );

      template<typename T> basic_format& operator%( T& arg );
      template<typename T> basic_format& operator%( const T& arg );

      string_t str() const;

      unsigned char exceptions( unsigned char newexcept );
      unsigned char exceptions() const;
  };

// 他にストリーム出力演算子が定義されています

typedef basic_format<char>     format;
typedef basic_format<wchar_t> wformat;
```

書式指定文字列、及び書式化用の引数を管理するクラスです。書式指定文字列の文法については、「書式指定」の項をご覧下さい。

```
template<class T> basic_format& operator%( T& arg );
template<class T> basic_format& operator%( const T& arg );
```

演算子%によって、引数をbasic_formatオブジェクトに渡します。複数個の引数を与える場合は、formatter % arg1 % arg2 % ...の形で複数個%演算子をつづけます。

```
string_t str() const;
```

整形済み文字列を、文字列型の値に変換して返します。

```
unsigned char exceptions( unsigned char newexcept );
unsigned char exceptions() const;
```

エラーが発生した際に例外を投げるか、単にその処理を終了するかを、エラーの種類毎にビットフラグで指定します。フラグと例外の詳細は以下の「例外処理」の項をご覧下さい。

```
typedef basic_format<char>     format;
typedef basic_format<wchar_t> wformat;
```

basic_formatテンプレートを、よく使われるcharとwchar_tで特殊化したものです。メンバ関数等についてはbasic_formatの解説を参照して下さい。

## 例外処理

| 必要なヘッダファイル | #include <boost/format.hpp> |
|---|---|
| 名前空間 | boost::io |

### 例外クラス

```
class format_error      : public std::exception;
class bad_format_string : public format_error;
class too_few_args      : public format_error;
class too_many_args     : public format_error;
class out_of_range      : public format_error;
```

例外に関係する定義は、boostではなくboost::io名前空間のなかで行われていることに注意が必要です。format_errorクラスは、ライブラリの投げる例外クラスの基底です。Boost.Formatに関する全ての例外を捕捉するためには、

```
catch( boost::io::format_error& e ) { ... }
```

と記述します。

bad_format_stringは、書式指定文字列が不正であったことを示します。too_few_argsは、文字列生成に必要な全ての引数が%演算子で与えられる前に、formatオブジェクトが文字列化されようとしたことを示します。too_many_argsは、書式指定で必要とされた個数よりも、%演算子で与えた引数の数が多すぎることを示します。out_of_rangeは、basic_formatの幾つかの内部メンバ関数において、引数インデックスが範囲外であったことを示します。

### 例外フラグ

```
enum format_error_bits
{
    bad_format_string_bit = 1,
    too_few_args_bit      = 2,
    too_many_args_bit     = 4,
    out_of_range_bit      = 8,
    all_error_bits        = 255, // 以上4つ全ての場合に例外を投げる
    no_error_bits         = 0,   // 以上4つ全ての場合、例外を投げない
};
```

これらのフラグを1つ以上basic_format::exceptionsメンバ関数に渡すことで、formatでのエラー発生時に例外を投げるか単に失敗するかを切り替えます。フラグはそれぞれ同名の例外と対応します。全ての例外をONにするマスクall_error_bitsとOFFにするマスクno_error_bitsが定義済みです。

## ユーティリティ関数

| 必要なヘッダファイル | #include <boost/format.hpp> |
|---|---|
| 名前空間 | boost::io |

### 文字列化

```
template<class CharT, class Traits>
  std::basic_string<CharT,Traits>
```

```
    str( const basic_format<CharT,Traits>& f );
```

str(f)とf.str()は同じ動作です。

### マニピュレータ適用

```
template<typename Manip1, typename T>
  unspecified_group_type group( Manip1, const T& );
template<typename Manip1, typename Manip2, typename T>
  unspecified_group_type group( Manip1, Mainp2, const T& );
...
// (10引数版まで定義されています)
```

書式指定による整形に加えて、iostreamライブラリのマニピュレータを適用したいときに用いるユーティリティ関数です。group関数の最後の引数に整形したい値を、前半の引数に適用するマニピュレータを指定して呼び出し、その返値をbasic_format::operator%へ渡します。例えば次のコードは、100という値を16進数表示します。

```
cout << format( "%1%" ) % group(hex, 100);
```

## 書式指定

Boost.Formatライブラリの書式指定文字列は、基本的にUnix98 Open-group printfの構文(http://www.opengroup.org/onlinepubs/7908799/xsh/fprintf.html)に従っています。これは、C++標準ライブラリのstd::printf関数の拡張になっています。すなわち、幾つかの特殊な例を除き、指定は次の形式で行います。[]で囲った部分は省略可能です。

```
% [ N$ ] [ flags ] [ width ] [ . precision ] type-char
```

▲Src. 1-28 書式指定の構文

N$フィールドには、整数と、それに続けて'$'文字を記述します。この書式指定が何番目の引数に適用されるものかの指定となります。省略された場合、先頭の引数から順に適用されます。ただし、一つの書式指定文字列の中では、N$は全て明示するか全て省略するかのどちらかしか許されません。同じ番号Nを持つN$フィールドが複数存在することは可能です。

flagsフィールドには、次のフラグ文字を指定します。複数個並べて指定することも可能です。

Tbl. 1-10　flagsフィールド

| フラグ文字 | 効果 |
|---|---|
| '-' | 左寄せ |
| '=' | 中央寄せ |
| '+' | 正の数に対し+符号を出力する |
| '#' | 8進、16進表記の際に、接頭辞0や0xを出力する |
| '0' | 余白は文字'0'で埋める |
| ' ' | 余白は空白文字' 'で埋める |

widthフィールドには、結果文字列の最低文字数を指定します。ここでの指定より整形結果が短い場合、flagsフィールドの指定に従って'0'や空白文字が埋められます。

precisionフィールドは、ドット文字に続けて数字を指定します。type-charが'e'または'f'で浮動小数点数を出力する場合には、小数点以下の桁数として解釈します。それ以外で浮動小数点数を出力する場合は、全体の桁数として解釈します。type-charが's'または'S'の時は、全体の文字数として解釈し、結果文字列の先頭部分のみを出力します。

type-charフィールドは必須です。以下のいずれか一文字を指定します。

Tbl. 1-11　type-charフィールド

| Type-Char文字 | 効果 |
|---|---|
| p, x | 16進出力 |
| o | 8進出力 |
| e | 浮動小数点数の場合、指数表記で出力 |
| f | 浮動小数点数の場合、固定小数点表記で出力 |
| g | 浮動小数点数の場合、デフォルトの表記方式で出力 |
| X, E, G | 効果はそれぞれx,e,gと同じだが、出力に大文字を使う |
| d, i, u | 10進出力 |
| s, S | 文字列として出力(precisionフィールドの説明参照) |

| | |
|---|---|
| c, C | 変換結果の最初の1文字のみ出力 |
| % | 文字'%'自体を出力(先頭の'%'と続けて"%%"と書いたとき) |

これらの表に無くてC++のprintf関数にはある書式指定文字は、単に無視されるようになっています。

また、以上の構文による指定と別に、若干の特別な書式が定義されています。Nには整数、cには一文字が入ります。

| | |
|---|---|
| %N% | %N$sと同じ効果。N番目の引数を文字列化する |
| %\| 書式指定 \| | %書式指定 と同じ意味。可読性や曖昧さの除去のためには、括弧として\|\|を使います |
| %Nt | 絶対位置指定。この書式指定の位置が結果文字列全体の第N文字目になるように' '文字を埋めて位置調整します |
| %NTc | 絶対位置指定。この書式指定の位置が結果文字列全体の第N文字目になるように、文字cを埋めて位置調整します |

# 1.7 String Algorithms(基本)

Boost C++ Libraries Programming　さまざまな文字列処理

C++言語は、PerlやRubyといったスクリプト言語と比較して、文字列処理能力が不足していると言われています。C++の標準ライブラリには文字列を表現するstd::stringクラスがありますが、確かに、std::stringでは文字列のコピーや結合などの基本的な操作のみしか提供されていません。Boost String Algorithmsライブラリでは、もっと様々な、よくある文字列処理が関数化されています。一例をあげると、以下のような関数があります。(全関数のリストは、P.86のリファレンスをご参照ください。)

- 文字列を特定の文字で分割：split
- 文字列の配列やvectorを特定の文字で結合：join
- 文字列の大文字化、小文字化：to_upper, to_lower
- 文字列前後の空白除去：trim, trim_left, trim_right
- 文字列が特定の文字列で始まっているか判定：starts_with
- 部分文字列を検索：find_first
- 部分文字列を置換：replace_first

例えば、文字列を特定の文字で分割するsplit関数と、文字列の配列やvectorを特定の文字で結合するjoin関数は、次のサンプルのように使います。

```
#include <iostream>
#include <string>
#include <vector>
#include <boost/algorithm/string.hpp>
using namespace std;
using namespace boost::algorithm;

int main()
{
    string message = "This is a pen";

    // 文字列messageを空白文字で区切ってvに格納
    vector<string> v;
    split( v, message, is_space() );
```

```
    // vの要素を"-"で結合して表示
    cout << join( v, "-" ) << endl;
}
```

▲Src. 1-29 string_algo1.cpp

実行結果

```
This-is-a-pen
```

split関数の第3引数には、区切りに使う文字を判別するオブジェクトを渡します。よく使われるのは、空白文字を意味するis_space()と、具体的に区切り文字を指定していくis_any_of()です。また、デフォルトでは区切り文字が連続していた場合に空文字列が結果に含まれますが、第4引数にtoken_compress_onを指定すると、複数並んだ区切り文字が一つの区切りと見なされるようになります。Src. 1-30が、これらの引数を使い分けた例です。

```
#include <iostream>
#include <string>
#include <vector>
#include <boost/algorithm/string.hpp>
using namespace std;
using namespace boost::algorithm;

int main()
{
    vector<string> v;

    // '-'か'+'の字で区切る
    split( v, "abc-def+ghi", is_any_of("-+") );
    cout << join( v, "<>" ) << endl;

    // 複数の空白で区切られている場合 (token_compress_off)
    split( v, "This  is  a  pen", is_space() );
    cout << join( v, "<>" ) << endl;

    // 複数の空白で区切られている場合 (token_compress_on)
    split( v, "This  is  a  pen", is_space(), token_compress_on );
    cout << join( v, "<>" ) << endl;
}
```

▲Src. 1-30 string_algo2.cpp

実行結果

```
abc<>def<>ghi
This<><>is<><>a<><>pen
This<>is<>a<>pen
```

次に別の例として、trim, all, ends_with の三つの関数を使ってみたものをお見せします。ユーザーから数字を入力してもらって、それが100の倍数になっているかどうか、言い換えれば最後が00で終わっているかどうかを判定するプログラムです。

```
#include <iostream>
#include <string>
#include <vector>
#include <boost/algorithm/string.hpp>
using namespace std;
using namespace boost::algorithm;

int main()
{
    for(;;)
    {
        cout << "数字を入れてね> ";

        // 1行読み込む
        string s;
        if( !getline(cin, s) )
            break;

        // nameの前後の空白を取り除く
        trim(s);

        // nameの文字が全て数字 && "00" で終わっているならば...
        if( all(s, is_digit()) && ends_with(s,"00") )
            cout << "100の倍数です" << endl;
    }
}
```

▲Src. 1-31 string_algo3.cpp

実行結果

```
数字を入れてね > 200
100の倍数です
数字を入れてね > 12
数字を入れてね >     34
数字を入れてね >     400
100の倍数です
```

ちょっと気にとめておいた方がいい点が二つほどあります。一つめは、trim関数をはじめとして、文字列を変更する系のアルゴリズム関数は、元の文字列を直接書き換えるということです。元の文字列には手を加えずに、変更後の文字列を新しく作って返して欲しいときは、_copyを付けた名前の関数を使います。例えばtrimされた新しい文字列が欲しい場合は、trim_copyです。

```
string trimmed = trim_copy(s);
```

もう一つ要注意なのは、ends_withなど検索系の関数は、残念ながらShift_JISやEUC-JPなどのマルチバイト文字に対応していない点です。検索の結果、文字の切れ目でないところがマッチしてしまう可能性があります。マルチバイト文字列を処理する場合は、UTF-8など元々この問題が起こりにくい文字コードの環境であることを確認してから使うか、wchar_t、wstringなどのワイド文字に変換してから全ての文字列処理をおこなうようにします。

```
// Shift_JISでは"表"の2バイト目が0x5c('¥¥'文字)なので問題となる
cout << ends_with( "表示",  "¥¥示") << endl; // true  : マルチバイト文字
cout << ends_with(L"表示", L"¥¥示") << endl; // false : ワイド文字ならOK
```

# 1.8 String Algorithms(検索・置換系)

Boost C++ Libraries Programming さまざまな文字列処理

文字列から部分文字列を取り出すアルゴリズムは、find_ではじまる名前となっています。find_系の関数は、結果をiterator_range型(P.407参照)で返します。iterator_rangeからは、beginとendメンバ関数をつかって、検索結果の範囲を取得できます。以下のサンプルでは、"Hello Hello Hello"という文字列から最初の"Hello"を検索しています。最初の出現を検索するにはfind_firstを使います。見つからなかった場合は、空のiterator_rangeが返ります。

```
#include <iostream>
#include <string>
#include <vector>
#include <boost/algorithm/string.hpp>
using namespace std;
using namespace boost;
using namespace boost::algorithm;

int main()
{
    string s = "Hello Hello Hello";
    iterator_range<string::iterator> r = find_first( s, "Hello" );

    cout << r.begin()-s.begin() << " - " << r.end()-s.begin() << endl;

    r = find_first( s, "GoodBye" );
    if( !r )
        cout << "Not Found" << endl;
}
```

▲Src. 1-32 string_algo4.cpp

実行結果

```
0 - 5
Not Found
```

検索して見つかった部分を別の文字列で置き換えるには、replace_ではじまる関数を使います。また、erase_ではじまる関数を使うと、見つかった部分を削除

することができます。replace_ や erase_ には、文字列を直接書き換える _copy なしのバージョンと、置き換え済み文字列を新しく返す _copy 付きが用意されています。

```
#include <iostream>
#include <string>
#include <vector>
#include <boost/algorithm/string.hpp>
using namespace std;
using namespace boost::algorithm;

int main()
{
    string s = "Hello Hello Hello";

    // 最後のHelloをHiに置き換え
    replace_last( s, "Hello", "Hi" );
    cout << s << endl;

    // Helloを全部削除した新しい文字列を返す
    cout << erase_all_copy( s, "Hello" ) << endl;

    // 最初のHelloをGoodByeに置き換えた新しい文字列を返す
    cout << replace_first_copy( s, "Hello", "GoodBye" ) << endl;
}
```

▲Src. 1-33 string_algo5.cpp

実行結果

```
Hello Hello Hi
  Hi
GoodBye Hello Hi
```

さて、ここまで紹介したサンプルソースは全てstd::stringに対して処理をおこなっていました。しかし実は、String Algorithmsのすべての関数は、iterator_rangeを含む任意の「範囲」オブジェクトに適用可能なように、一般的に定義されています。つまり、find_系関数の返値をさらにString Algorithms関数で処理することができます。いくつか使用例をご覧ください。

```
#include <iostream>
#include <string>
```

```
#include <vector>
#include <boost/algorithm/string.hpp>
using namespace std;
using namespace boost::algorithm;

int main()
{
    string s = "Hello Hello Hello";

    // 2番目のHelloを大文字にする
    //（find_nthは0始まりでカウントするので引数は1）
    to_upper( find_nth(s, "Hello", 1) );
    cout << s << endl;

    // 先頭6文字を取り出してきて、両端の空白除去
    cout << "<" << trim_copy( find_head( s, 6 ) ) << ">" << endl;
}
```

▲Src. 1-34　string_algo6.cpp

実行結果

```
Hello HELLO Hello
<Hello>
```

ただし、_copyの付かない方のtrimやreplace関数のように、直接引数を書き換える関数は、一般の範囲には適用できません。

Boost C++ Libraries Programming さまざまな文字列処理

# Reference

## 基本関数

| 必要なヘッダファイル | #include <boost/algorithm/string.hpp> |
|---|---|
| 名前空間 | boost::algorithm |

### 大文字・小文字変換

```
void to_upper( Range, const std::locale& loc = std::locale() );
void to_lower( Range, const std::locale& loc = std::locale() );

template<typename OutIter>
  OutIter to_upper_copy( OutIter it, Range,
                         const std::locale& loc = std::locale() );
 Sequence to_upper_copy( Sequence,
                         const std::locale& loc = std::locale() );

template<typename OutIter>
  OutIter to_lower_copy( OutIter it, Range,
                         const std::locale& loc = std::locale() );
 Sequence to_lower_copy( Sequence,
                         const std::locale& loc = std::locale() );
```

*Range*と書かれた引数には、std::stringやiterator_rangeなどの、beginとendのとれるRangeオブジェクトを指定できます。*Sequence*と書かれた引数には、std::stringなどのシーケンスオブジェクトを指定できます。

指定された文字列の中のアルファベットを、すべて大文字(to_upperの場合)か小文字(to_lower)に変換します。_copyの付いたバージョンは、引数を直接書き換えるのではなく、新しい文字列を作って返します。最後の引数にlocaleを指定すると、アルファベットや大文字・小文字の判定をそのlocaleを使っておこなうようになります。

## 空白除去

```
void trim      ( Sequence, const std::locale& loc = std::locale() );
void trim_left ( Sequence, const std::locale& loc = std::locale() );
void trim_right( Sequence, const std::locale& loc = std::locale() );

Sequence trim_copy( Sequence, const std::locale& loc = std::locale() );
template<typename PredicateT>
  void trim_if( Sequence, PredicateT isSpace );
template<typename PredicateT>
  Sequence trim_copy_if( Sequence, PredicateT isSpace );
template<typename OutIter, typename PredicateT>
  OutIter trim_copy_if( OutIter it, Range, PredicateT isSpace );

// trim_left, trim_right についても _copy, _if, _copy_if バージョンがあります
```

文字列の左端(trim_left)、右端(trim_right)、または両側(trim)の空白文字を取り除きます。_copy版は渡された文字列を書き換えずに新しい文字列を返します。_if版は、文字を渡すとboolを返す関数オブジェクトを引数に渡すと、その関数オブジェクトがtrueになる文字を両端から取り除きます。

## 包含判定

```
bool starts_with( Range heystack, Range needle );
bool   ends_with( Range heystack, Range needle );
bool    contains( Range heystack, Range needle );

bool                  equals( Range lhs, Range rhs );
bool lexicographical_compare( Range lhs, Range rhs );

template<typename PredicateT>
  bool  starts_with( Range heystack, Range needle, PredicateT Comp );
  bool istarts_with( Range heystack, Range needle );
// end_with, contains, equals, lexicographical_compare についても
// テンプレート版と頭にiのついたバージョンがあります
```

starts_withは、文字列heystackの中の先頭が文字列needleになっているかどうかを判定します。ends_withは、heystackの最後がneedleになっているかどうかを判定します。containsは、文字列中のどこかにneedleがあるかどうかを判定します。istarts_withのように関数名がiで始まっているものは、大文字

小文字の違いを無視して判定します。また、引数の最後に、二つcharをとって「等しい」文字かどうかを返す関数オブジェクトを指定することも可能です。

equalsは、二つの文字列が等しいかどうかを判定します。文字列がstd::stringの場合はoperator==と同値です。lexicographical_compareは辞書順での前後関係を判定します。std::stringの場合はoperator<と同値です。どちらも、iequals, ilexicographical_compareという大文字小文字の違いを無視する関数が用意されています。

## 文字種検査

```
template<typename PredicateT>
  bool all( Range, PredicateT pred );
```

文字列の全ての文字が関数オブジェクトpredに対してtrueとなる場合に限り、trueを返します。

```
PredicateT  is_space(const std::locale& Loc = std::locale());
PredicateT  is_alnum(const std::locale& Loc = std::locale());
PredicateT  is_alpha(const std::locale& Loc = std::locale());
PredicateT  is_cntrl(const std::locale& Loc = std::locale());
PredicateT  is_digit(const std::locale& Loc = std::locale());
PredicateT  is_graph(const std::locale& Loc = std::locale());
PredicateT  is_lower(const std::locale& Loc = std::locale());
PredicateT  is_print(const std::locale& Loc = std::locale());
PredicateT  is_punct(const std::locale& Loc = std::locale());
PredicateT  is_upper(const std::locale& Loc = std::locale());
PredicateT is_xdigit(const std::locale& Loc = std::locale());

PredicateT is_classified(std::ctype_base::mask Type,
                         const std::locale& Loc = std::locale());
```

all関数やtrim_if関数、split関数などに渡せる関数オブジェクトを作成するための関数です。それぞれ、関数名に対応する文字種の文字ならばtrue、それ以外にはfalseを返すような関数オブジェクトを返します。is_classified関数は、標準ライブラリの<cctype>ヘッダで定義された文字種マスクを受け取って、対応する関数オブジェクトを返します。

## 検索

```
Range  find_first( Range heystack, Range needle );
Range  find_last ( Range heystack, Range needle );
Range  find_nth  ( Range heystack, Range needle, int n );
Range ifind_first( Range heystack, Range needle );
Range ifind_last ( Range heystack, Range needle );
Range ifind_nth  ( Range heystack, Range needle, int n );

template<typename SeqSeq>
  void find_all( SeqSeq& result, Range heystack, Range needle );

Range find_head ( Range heystack, int n );
Range find_tail ( Range heystack, int n );

template<typename PredicateT>
  Range find_token( Range heystack, PredicateT pred,
    token_compress_mode_type eCompress = token_compress_off );

Range find_regex( Range heystack, const regex& re,
                  match_flag_type Flags = match_default );
```

文字列heystackの中から、指定された条件の部分文字列のある範囲を返します。見つからなかった場合は空の範囲を返します。

find_first, find_last, find_nthはそれぞれ、指定された文字列needleの最初、最後、n番目の出現を検索します。ifind_firstのようにifind_で始まる関数は、大文字小文字の違いを無視して検索をおこないます。

find_allは、部分文字列needleの出現を先頭から順にすべてresultに追加して返します。resultには、std::vector<iterator_range<...>>やstd::vector<string>などのコンテナを指定します。

find_head, find_tailは、それぞれ先頭、末尾のn文字を返します。文字列全体がn文字より短い場合は、文字列全体を返します。

find_tokenは、第二引数で指定された文字種の文字の最初の出現を返します。第3引数としてtoken_compress_onを指定すると、第二引数で指定された文字種の文字からなる文字列の最初の出現を返します。例えば、find_token("abc123def", is_digit(),token_compress_on)は、"123"を返します。

find_regexは、Boost.Regexの正規表現オブジェクトを受け取ってそれにマッチする範囲を返します。

## 置換

```
    void replace_first( Sequence heystack, Range needle, Range format )
Sequence replace_first_copy( Sequence heystack, Range needle,
                             Range format )
template<typename OutIter>
  OutIter replace_first_copy( OutIter it, Range heystack, Range needle,
                              Range format )

    void ireplace_first( Sequence heystack, Range needle, Range format )
Sequence ireplace_first_copy( Sequence heystack, Range needle,
                              Range format )
template<typename OutIter>
  OutIter ireplace_first_copy( OutIter it, Range heystack, Range needle,
                           Range format )

    void erase_first( Sequence heystack, Range needle )
Sequence erase_first_copy( Sequence heystack, Range needle )
template<typename OutIter>
  OutIter erase_first_copy( OutIter it, Range heystack, Range needle )

    void ierase_first( Sequence heystack, Range needle )
Sequence ierase_first_copy( Sequence heystack, Range needle )
template<typename OutIter>
  OutIter ierase_first_copy( OutIter it, Range heystack, Range needle )

// _firstだけでなく、
// replace_last
// replace_nth
// replace_head
// replace_tail
// replace_token
// replace_regex
// replace_all
// およびそれぞれのerase版、_copy版も定義されています。
```

replace_系関数は、文字列heystackの中で指定された部分を、文字列formatで置き換えます。検索方法の指定は対応するfind_系関数と同じで、置き換え後の文字列を最後の引数として指定します。erase_系関数は、検索で見つかった部分を削除します。replace_( ..., "" )と同じ効果があります。

### 分割

```
template<typename SeqSeq, typename PredicateT>
  void split( SeqSeq& result, Range heystack, PredicateT saparator,
    token_compress_mode_type eCompress = token_compress_off );
```

文字種 separator の文字で、文字列 heystack を分割して、分割結果をコンテナ result に追加して返します。

### 連結

```
template<typename SeqSeq>
  Range join( const SeqSeq& seq, Range separator );
```

文字列のシーケンス seq の全要素を、文字列 separator を間に挟みながら一つの文字列へと結合します。

## 汎用検索処理

| 必要なヘッダファイル | #include <boost/algorithm/string.hpp> |
|---|---|
| 名前空間 | boost::algorithm |

### 汎用検索関数

```
template<typename FinderT>
  Range find( Range str, FinderT finder );
```

文字列 str から、条件を満たす部分文字列を検索して返します。条件は Finder コンセプトを満たすオブジェクトとして指定します。

### Finderコンセプト

以下のメンバ関数一つを持つオブジェクトは、検索条件としてfind関数の第2引数に渡すことができます。operator()では、範囲[s,e)から、条件を満たす部分範囲を検索して返します。

```
class
{
public:
    template<typename FwdIter>
      iterator_range<FwdIter,FwdIter> operator()( FwdIter s, FwdIter e );
};
```

### 定義済みFinder生成関数

```
Finder first_finder( Range needle );
Finder  last_finder( Range needle );
Finder   nth_finder( Range needle, int n );
Finder  head_finder( int N );
Finder  tail_finder( int N );
Finder token_finder( PredicateT pred,
  token_compress_mode_type eCompress = token_compress_off );
Finder regex_finder( const regex& re, match_flag_type Flags
                     = match_default
```

それぞれ同じ名前のfind_系関数と同じ働きをするFinderを生成して返します。

## 汎用置換処理

| 必要なヘッダファイル | #include <boost/algorithm/string.hpp> |
|---|---|
| 名前空間 | boost::algorithm |

## 汎用置換関数

```
template<typename FinderT, typename FormatterT>
  void find_format( Sequence str, FinderT finder, FormatterT formatter );
```

文字列strから、条件を満たす部分文字列を検索して、見つかった部分を置換して返します。条件はFinderコンセプトを満たすオブジェクトとして指定します。置換方法はFormatterコンセプトを満たすオブジェクトで指定します。

## Formatterコンセプト

Finderの検索結果を受け取って、それを整形して文字列として返す関数オブジェクトが、Formatterです。

```
class
{
public:
    Range operator()( Finder::operator()の返値型 );
};
```

## 定義済みFormatter生成関数

```
Formatter    const_formatter( Range fmt );
Formatter    empty_formatter();
Formatter identity_formatter();
Formatter    regex_formatter( const std::string& fmt );
```

const_formatterは、常に同じ文字列fmtを返すFormatterです。empty_formatterは、常に空文字列を返します。identity_formatterは、Finderの結果をそのまま変更せずに返します。regex_formatterはregex_finderと常に対にして使います。Boost.Regexの正規表現置換の形式で置換文字列を指定します。

## イテレータ

| 必要なヘッダファイル | #include <boost/algorithm/string.hpp> |
|---|---|
| 名前空間 | boost::algorithm |

### find_iterator

```
template<typename Iter>
  class find_iterator
  {
      find_iterator();
      template<typename FinderT>
        find_iterator(Iter, Iter, FinderT);
      template<typename FinderT>
        find_iterator(Range, FinderT);
  };
```

指定したFinderにマッチする部分を先頭から順に列挙するイテレータです。

### split_iterator

```
template<typename Iter>
  class split_iterator
  {
      split_iterator();
      template<typename FinderT>
        split_iterator(Iter, Iter, FinderT);
      template<typename FinderT>
        split_iterator(Range, FinderT);
  };
```

指定したFinderにマッチ「しない」部分を先頭から順に列挙するイテレータです。