俺 Tokenizer を作る
～Boost.Tokenizer のカスタマイズ～

tt_clown (津川　知朗）
tt.clown@gmail.com
http://d.hatena.ne.jp/tt_clown/

自己紹介

- tt_clown（津川　知朗）
  - ▶tt_ プレフィクスは要らない子
  - ▶小さなベンチャー会社でコードを書いてます
- 公開しているもの
  - ▶CLX C++ Libraries: http://clx.cielquis.net/
    - FAQ: Q.名前被ってね？ A. ゴメンナサイ
- Blog
  - ▶Life like a clown: http://d.hatena.ne.jp/tt_clown/

目次

- 今日の目標: scanf () 風の機能を持つクラスの実装
- 実装指針: Boost.Tokenizer をカスタマイズする
- Boost.Tokenizer のポリシー・クラス
- FormatSeparator
  - ▶実装方針
  - ▶サンプル・コード
- Scanner クラスの実装
- まとめ

今日の目標

- scanf() のような機能を持つクラス scanner を作る
  - ▶http://clx.cielquis.net/scanner.html

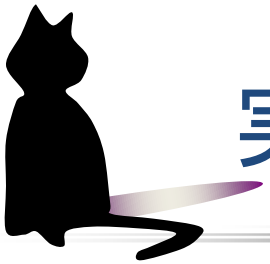
```
int main() {
    std::string s = "2009/12/12T13:10:25"
    std::string format = "%s/%s/%sT%s:%s:%s";

    int year = 0, mon = 0, day = 0;
    int hour = 0, min = 0, sec = 0;

    scanner(s, format) % year % mon % sec % hour % min % sec;

    // 結果を表示するためのコードを書く.
    return 0;
}
```

このクラスを実装する



実装の指針

- Boost.Tokenizer をカスタマイズ
  - ►ポリシー・クラスの自作によるカスタマイズの一例
- ポリシー・クラスとは？
  - ►Modern C++ Design [1] で広まった設計思想
  - ►あるクラスの鍵となる「動作」に Interface を決めておく
    - この Interface に合致するクラスがポリシー・クラス
- ポリシー・クラスのメリット
  - ►状況に応じて「動作」を選択できるため拡張性が高い

[1] アンドレイ・アレキサンドレスク（訳：村上 雅章），“Modern C++ Design”，2001.

Boost.Tokenizer のポリシー・クラス

Boost.Tokenizer の宣言

```
template <
    class TokenizerFunc,
    class Iterator = std::string::const_iterator,
    class Type = std::string
>
class tokenizer;
```

ここにポリシー・クラスを指定する

TokenizerFunc の Interface

```
class tokenizer_func_skelton {
public:
    template <class InIter, class Token>
    bool operator()(InIter& next, InIter last, Token& dest);

    void reset();
};
```

[*next*, *last*) から次のトークンを切り出して *dest* へ格納する.
切り出しに成功した場合は true, それ以外は false を返す.

FormatSeparator

- scanf() 風の文字列分割を実現するための Boost.Tokenizer のポリシー・クラス

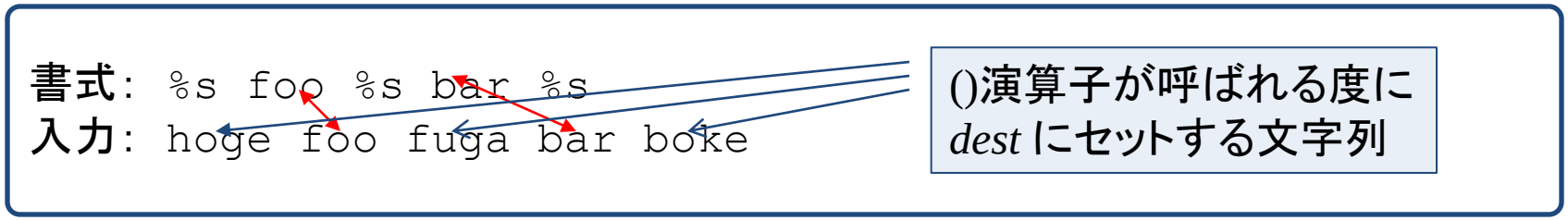
```
class format_separator {
public:
    format_separator(const string_type& fmt, bool x = true) :
        fmt_(fmt), skipws_(x), cur_(fmt_.begin()) {}

    void reset() { ... }

    template <class InIter, class Token>
    bool operator()(InIter& next, InIter last, Token& dest) {
        ....
    }
};
```

FormatSeparator 実装方針

- 変換指定*1以外の文字列マッチで判定



- 型修飾子は s のみ (%d, %f, %x, などを許さない)
  - ▶型変換は，scanner に任せる
    - 代入直前に lexical_cast を用いて変換する
- 空白文字の扱いを scanf() に似せる

*1 ``%'' で始まる文字列

FormatSeparator サンプルコード

```
int main() {
    std::string s = "Sat Dec          12 13:10:25 JST 2009";
    std::string format = "%s %s %s %s:%s:%s %s %s";

    typedef boost::tokenizer<format_separator> fmttokenizer;
    format_separator sep(format);
    fmttokenizer token(s, sep);

    std::cout << "source: " << s << std::endl;
    for (fmttokenizer::iterator pos = token.begin();
        pos != token.end(); ++pos) {
        std::cout << "<" << *pos << "> ";
    }
    std::cout << std::endl;
    return 0;
}
```

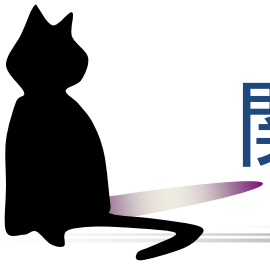
Scanner の実装

```
class scanner {
public:
    scanner(const string_type& s, const string_type& fmt) :
        v_(), cur_() {
        format_separator sep(fmt);
        boost::tokenizer<format_separator> x(s, sep);
        v_.assign(x.begin(), x.end());
        cur_ = v_.begin();
    }

    template <class Type>
    scanner& operator%(Type& dest) {
        if (cur_ != v_.end()) {
            dest = boost::lexical_cast<Type>(*cur_++);
        }
        return *this;
    }
};
```

まとめ

- scanf() 風の機能を持つクラスの実装
- Boost.Tokenizer のカスタマイズと言うアプローチ
  - ►ポリシー・クラスの作成/利用方法の一例として紹介
- ポリシーに基づくクラス設計のメリット
  - ►鍵となる「動作」を Interface として抽象化しておく
    - それ以外の部分の再利用性が高まる
  - ►状況に応じたポリシー・クラスを選択できる
    - 高いカスタマイズ性を確保できる



関連URL

- clx::scanner
  - ▶http://clx.cielquis.net/scanner.html
- clx::tokenizer_func
  - ▶http://clx.cielquis.net/tokenizer_func.html
- boost::tokenizer で scanf を作ってみる
  - ▶http://d.hatena.ne.jp/tt_clown/20090902/1251822236