

作者：八方齋

E-mail：[make.sense@msa.hinet.net](mailto:make.sense@msa.hinet.net)

**P.S 歡迎轉貼但請註明出處！**

## 緣起：

C++是一種強大無比的語言，但是有一樣功能卻是從 C 語言時代就一直被人詬病，那就是「字串」。當然，有些朋友認為 `sprintf`, `sscanf` 等等就已經夠用了，甚至到了 C++ 時代，這些函數仍是程式員的最愛（包括在下），但是不可諱言的，這些函數的確有些跟不上時代（這些東西起碼有 2X 年的歷史了）。

首先，這些函數都缺乏 `type-safe` 的特性，因為這些函數的操作幾乎都依靠格式化字串，僅有的除錯資訊就是這些函數 `run-time` 時的 `return` 值，但是這些函數的 `return` 值只有告訴你成功轉換了幾個引數，卻沒有告訴你轉換失敗的是哪幾個，甚至應該轉換失敗的也轉換成功（如 `float` 不小心轉成 `int`），這些也許都還可以歸咎程式員自己寫程式不夠細心，但是還有一個更糟糕的問題是，格式化字串的語法並非放諸四海而皆準。

**(感謝 `cszone xam` 網友指正，此段已修改-2006/3/19)**

以 `sscanf` 為例，讓我們思考下面這段程式：

```
char test_str[]="aaa,bbb";
```

```
sscanf(test_str, "%s,%s", p1, p2); //p1, p2 is char*
```

你會發現引數 `p2` 不會得到 "bbb"，因為 `sscanf` 是以 `space` 做為分界，所以 `p1` 會得到 "aaa,bbb"，而 `p2` 什麼也得不到；假如 `test_str="aaa, bbb"` (add space)，此時 `p1="aaa,"`，`p2="bbb"`，但仍不是我們想要的 `p1="aaa"`、`p2="bbb"`。老練的 C 程式員的解法是改用 `strtok` 將字串分解，這在 C 語言中是合理的解法，但是在 C++ 中將 `std::string::c_str()` 丟給 `strtok()` 執行將是一場浩劫，原因是 `strtok` 對字串通常採取破壞的手段（填入 0 進行分隔）。

## Regular Expression

讓我們看看另一種語言 Perl，Perl 眾所皆知對字串處理的能力超強，而程式可與 Regular Expression[1] 溶為一體，而 Regular Expression（後面簡稱 `regex`）更不用說了，熟習 Linux 操作的人很少沒有人用過它（`grep`, `sed`...），根據你所下的 `expression`，可以很方便的對文字進行精確的 `search/replace`；不要以為這樣的功能只存在於 `script language`，事實上 `regex` 已經是 UN\*X 標準配備（POSIX），連 C 語言也可以享用到！

就算在 windows 環境下，我們也有免費的 C regex lib 可用，PCRE 就是十分知名的一套，大家常用的 php, python 等都有引用做為核心元件之一，不過 PCRE 本身並不好操作，參數多是一個原因，重點是 PCRE 操作的仍是 C-style 字串，而非 C++ 程式員熟習的 std::string，當然我們可以透過一些包裝的手段來使得 PCRE 也可用在 std::string 之上；但是有了 Regex++，我們就何苦重新發明輪子？

## Regex++

Regex++[2]是一套完全以 C++ 寫成的 regex lib，支援 VC、BCB、GCC 等等 C++ compiler，由於使用 template 的方式設計，於是不但可以用在 std::string 上，char\*、甚至是 stlport 特有的 rope 都沒問題，聰明的你一定已經猜到，沒錯，他也是利用 iterator 的方式對字串操作，如此一來你在 STL 學到的知識仍然能用在 Regex++ 身上，讓我們看看下面這個簡單的例子[3]：

```
bool validate_card_format(const std::string& s)
{
    static const boost::regex e("\\d{15,16}");
    return regex_match(s,e);
}
```

這個函數的用途是檢查字串長度是不是 15or16 個字元，並且每個字元都必須是數字(0~9)，沒記錯的話這是用來檢查信用卡號碼的格式正不正確，\\d 代表必須為數字，{15,16}則限制了長度必須為 15~16 之間。

現在我們討論一個更複雜的例子：

```
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)
{
    std::string result = regex_merge(s, e, machine_format, boost::match_default |
    boost::format_sed | boost::format_no_copy);
```

```

    if(result.size()==0)
        throw std::runtime_error("string is not a credit card number");
    return result;
}

std::string
human_readable_card_number(const std::string& s)
{
    std::string result = regex_merge(s, e, human_format, boost, boost::match_default
| boost::format_sed | boost::format_no_copy);

    if(result.size()==0)
        throw std::runtime_error("string is not a credit card number");
    return result;
}

```

machine\_readable\_card\_number()的功能是把信用卡號碼轉成適合機器讀取的格式，比方說 XXXX-XXXX-XXXX-XXXX 是我們一般人習慣的格式，假設我們想把它存進 database 做為索引，以字串作為欄位格式似乎不是個好主意（效能比較差），所以你可以利用 machine\_readable\_card\_number()進行轉換，然後再用我們的老朋友 atoi()或是 istringstream 轉成整數存入資料庫中。同樣的，為了讓 user 心情愉快，當我們從 database 中讀出這個整數時，可以利用 human\_readable\_card\_number()轉成一般人習慣的格式J

這裡大略解釋一下程式內容（假如你學過 Perl，可以跳過這段），regex\_merge 的用途是「當字串符合 pattern 時，依據 format 進行合併」，pattern 指的是第二個參數，format 是三個參數，至於旗標暫時不用去管他，pattern 中的\A 與\z 限制字串必須全部 match，而不是片段，第一個小括號(\d{3,4})對應到 format 中的\1(以此類推)，[]代表一個集合，[- ]代表“-“與 space，加上”?”代表“-“與 space 只能有一個或零個。

## Overloaded Operator >>

如果剛剛舉的例子還不能讓你感受到 regex 的強大威力，那接下來的範例肯定會讓你耳目一新，大家都知道 istringstream 的用法如下：

```

int a, b;
string str="123 567";

```

```
istringstream iss(str);
iss>>a>>b;
```

很顯然易見，istringstream 是以空白作為區隔讀取字串，假設我們今天不想用空白而想用”，”呢？雖然 istringstream 也能設定 format state 讓我們作一些小改變，不過要是逗號旁多了一些空白，istringstream 也能輕鬆處理嗎（假設這個字串是從某個設定檔中讀出，這個設定檔是由 user 自行輸入產生的）？起碼以筆者現在的 C++功力，是無法簡單用 istringstream 完成的。

但是我們現在有了 Regex++這個強力的武器，問題就簡單多了，Regex++有個 template function - regex\_search()可以對 string 進行搜尋，之前曾經提到可以要求 Regex++對字串的比對是「完全吻合」，當然也可以要求 Regex++「部分吻合」，這樣我們便可以設計一個 pattern 來搜尋以逗號作為分隔資料的字串，也許這樣講還是很模糊，我們來看看實際的程式：

```
typedef pair<string::const_iterator, string::const_iterator> ConstStdStrPtrPair;
```

```
template<class T>
ConstStdStrPtrPair operator>>(ConstStdStrPtrPair lhs, T& rhs)
{
    boost::match_results<std::string::const_iterator> what;
    static boost::regex expression(",* *([^\s]+) *,*");
    unsigned int flags = boost::match_default;

    if(boost::regex_search(lhs.first, lhs.second, what, expression, flags) &&
lhs.first!=lhs.second)
    {
        string buf(what[1].first, what[1].second);
        istringstream ss(buf);
        if( !(ss>>rhs) )
            throw runtime_error("Convert Fail!");
        lhs.first = what[0].second;
    }
    else
        throw runtime_error("Format Error or String End");
    return lhs;
}
```

```

template<class T>
ConstStdStrPtrPair operator>>(const std::string& lhs, T& rhs)
{
    boost::match_results<std::string::const_iterator> what;
    static boost::regex expression(",* *([^\s]+) *,*");
    unsigned int flags = boost::match_default;
    ConstStdStrPtrPair ptrPair;

    ptrPair.first = lhs.begin();
    ptrPair.second = lhs.end();

    if(boost::regex_search(ptrPair.first, ptrPair.second, what, expression, flags) &&
ptrPair.first!=ptrPair.second)
    {
        string buf(what[1].first, what[1].second);
        istream ss(buf);
        if( !(ss>>rhs) )
            throw runtime_error("Convert Fail!");
        ptrPair.first = what[0].second;
    }
    else
        throw runtime_error("Format Error or String End");
    return ptrPair;
}

```

我先解釋第二個 `operator>>`，第一個參數是 `string`，第二個是欲取得的值，將字串的開頭跟結尾傳入 `regex_search()` 進行搜尋，若是符合格式，`what` 會指向符合格式的子字串，`what[0].second` 會指向符合格式的子字串的下一個 `iterator`，可以作為下一次搜尋的起點，所以我把它與字串結尾存入一 `pair` 作為回傳值，然後把這個 `pair` 當成第一個 `operator>>` 的第一個參數，這樣從字串讀出資料時就如同 `istream` 一樣簡單：

```

//example
int a,b,c;
string testString="123,456,789";
testString>>a>>b>>c;

```

至於 `what[1].first` 與 `what[1].second` 可以很明白的看出是指向符合格式的子字串的

開頭與結尾，這邊的作法是另外存成字串，透過 `istringstream` 轉出，最後檢查有沒有轉換成功。

## 心得：

如何？`Rgex++`是不是替你的 `C++` 程式設計生涯帶來了曙光？其實剛剛的例子還有改進的空間，最顯而易見的就是 `pattern` 是寫死在 `operator>>` 裡，有沒有辦法讓 `user` 自行決定？或是「另外存成字串，透過 `istringstream` 轉出」這樣的動作會不會造成記憶上額外的開銷？有沒有什麼降低成本的辦法？這些就留給讀者去研究吧（假如你有什麼好點子請聯絡我，謝謝！）

事實上，作者已經利用 `Regex++` 完成了不少工作，例如 `html mail`（透過 `indy` 發信，`regex++` 則用來搜尋 `html` 中的影像檔並修改為 `outlook` 可讀取的格式），自訂的設定檔讀取模組（like `INI`），`parse C` 語言函數參數的型別與參數名稱...就是因為它這麼好用，才有感而發覺得該寫篇文章推廣一下，希望對大家的平日的工作能有所幫助J

## 參考資料：

[1] 學習 Regular Expression，國內就有幾個不錯的網站：

<http://www.cyut.edu.tw/~ckhung/olbook/gnulinux/regexp.shtml>

這是朝陽科技大學資管系洪朝貴老師寫的，內容有點亂，不過用 `Perl` 配合著練習會好很多。

<http://phi.sinica.edu.tw/aspac/reports/94/94019/>

中央研究院計算中心寫的一份簡介

<http://main.rtfiber.com.tw/~changyj/>

事實上，國內我還沒看過比龍門少尉寫的更淺顯易懂的（強力推薦）

[2] [http://ourworld.compuserve.com/homepages/John\\_Maddock/regexp.htm](http://ourworld.compuserve.com/homepages/John_Maddock/regexp.htm)

[3] 這兩個例子取自 `Regex++` 作者在 `Dr. Dobb's Journal October 2001` 發表的文章 "Regular Expressions in C++"，個人認為這篇文章是個很好的入門，`Regex++` 本身的範例實在有點嚇人。