



U.S. DEPARTMENT OF
ENERGY

PNNL-22590

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

The RE_c Utility Class

DES-0008

Revision 2

Charlie Hubbard
July 2010



Pacific Northwest
NATIONAL LABORATORY

*Proudly Operated by **Battelle** Since 1965*

The RE_c Utility Class

Charlie Hubbard

DES-0008
Revision 2
July 2010

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor Battelle Memorial Institute, nor any of their employees, *makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.* Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or Battelle Memorial Institute. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

PACIFIC NORTHWEST NATIONAL LABORATORY

operated by

BATTELLE

for the

UNITED STATES DEPARTMENT OF ENERGY

under

Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831-0062

ph: (865) 576-8401

fax: (865) 576-5728

email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161

ph: (800) 553-6847

fax: (703) 605-6900

email: orders@ntis.fedworld.gov

online ordering: <http://www.ntis.gov/ordering.htm>

Contents

1	Introduction	1
2	The RE_c Class	2
2.1	Public Methods	2
2.1.1	The Class Constructor	2
2.1.2	The Match() Method	3
2.1.3	The NumSubstrings() Method	4
2.1.4	The Dump() Method	4
2.2	Private Methods	5
2.2.1	The CompilePattern() Method	5
2.2.2	The ConvertPercentString() Method	5
2.3	Substring Access Operators	5
2.3.1	The [] Operator	5
2.3.2	The () Operator	6

1 Introduction

In many, many places throughout the code base, there is a need to parse strings. Quite often the parsing is most painlessly accomplished with regular expressions. The Perl scripting language offers a particularly powerful regular expression capability, and the Perl regular expression syntax is popular and well known among software developers. The *Perl Compatible Regular Expressions* (PCRE) library, is an open-source C library, widely available and supported on many different operating systems, that implements regular expressions using the Perl regular expression syntax. Unfortunately, as a straight C library, it is a bit awkward to work with.

This document describes the `RE_c` utility class, which functions as a C++ wrapper around the PCRE C library and smooths out some of the interface kinks the bare PCRE library has.¹ The `RE_c` class is defined in *re.h* and implemented in *re.cpp*. To work, it requires that the PCRE C library be installed and available on the system. On Ubuntu Linux systems, this amounts to installing the *libpcre3-dev* package.

The syntax for constructing Perl-type regular expressions is arcane, and regular expression strings can be difficult to decipher even for experienced users. While the construction of Perl type regular expressions is beyond the scope of this document, readers are referred to the Perl documentation website at <http://perldoc.perl.org/perlre.html> for further information. Information on the underlying PCRE C library can be found at <http://www.pcre.org>.

¹In more recent versions of the PCRE library, a C++ wrapper class is included as part of the library. However, to be compatible with our pre-existing and extensive software code base, we continue to use our `RE_c` wrapper class.

2 The RE_c Class

A slightly decluttered version of the RE_c class definition appears below. The full definition can be found in *re.h*.

```
class RE_c {
public:
    explicit RE_c(std::string const & pattern, int maxSubstrings = 30);
    ~RE_c();
    int Match( std::string const & source );
    int NumSubstrings();
    void Dump();
    const char* operator[](int ndx);
    std::string const operator()(int index);

    // some useful predefined regular expression patterns
    static std::string const TRIM;
    static std::string const COMMENT;
    static std::string const FP;
    static std::string const HEX;

private:
    void CompilePattern( std::string const & pattern );
    static std::string ConvertPercentString( std::string const & pattern );

    RE_c operator=( RE_c const & rhs);
    RE_c( RE_c const & rhs);
    int ssCount;
    int const offsetsSize;
    const char **ss;
    void *re;
};
```

2.1 Public Methods

2.1.1 The Class Constructor

```
explicit RE_c(std::string const & pattern, int maxSubstrings = 30);
```

The constructor takes two parameters.

- **pattern** — This is the regular expression used when comparing strings processed with the constructed object. The regular expression should adhere to standard Perl syntax for regular expressions with one exception: in Perl, many special codes for identifying certain groups of characters (like “\d” for any digit or “\s” for any whitespace character) are prefixed with a backslash character (\). In C and C++, the backslash character has special meaning inside string literals, and must itself be escaped with another backslash if it is to be treated literally. This duplication of backslashes rapidly becomes cumbersome. For that reason, the RE_c wrapper class uses the percent sign (%) instead of a backslash (so “%d” for any digit, “%s” for any whitespace character,

and so on). If a literal percent sign is required, it can be escaped with itself (that is to say, “%%” will be treated as a single, literal “%” in the regular expression).

- **maxSubstrings** — This contains the maximum number of matched substrings the regular expression will save. This parameter defaults to 30, which is more than enough for most situations. However, the user can override this if necessary.

The constructor simply passes the **pattern** string to the PCRE library to be *compiled*. That is to say, the pattern string is converted to an internal state machine representation (a nondeterministic finite automaton), which is run against the user’s comparison strings to find matches. Once an **RE_c** object has been instantiated, there is no way to change its regular expression pattern.

During compilation of the regular expression pattern, it is possible that an error can occur (if the pattern is malformed, for instance). If there are problems with the compile, the constructor throws an exception of type **std:string**, that contains a description of the problem.

2.1.2 The Match() Method

```
int Match(std::string const &source);
```

The **Match()** method takes one STL string parameter, **source**, that contains a string to be compared against the regular expression to see if it matches the regular expression’s pattern.

The **Match()** method returns one of the **re_ERR*** codes defined at the top of *re.h*. Typically, the caller will be looking either for **re_ERR_NO_ERROR** (meaning the string matched the pattern) or **re_ERR_NO_MATCH** (the string did not match the pattern). However, other error results are possible, as defined in the table below.

Error Code	Meaning
re_ERR_NO_ERROR	The string was matched by the regular expression with no problems
re_ERR_SUBSTRINGS_TRUNCATED	The string was matched by the regular expression, but there was not room to store all the extracted substrings
re_ERR_NO_MATCH	The string did not match the regular expression
re_PCRE_ERROR_BADMAGIC	The internal representation of the regular expression is corrupted
re_ERR_REGEX_FAILED	This is a catch-all error that gets returned when no other error code is more appropriate
re_ERR_EXTRACT_FAILED	The string matched, and there was room for the extracted substrings, but a problem occurred when requesting the substring list

2.1.3 The NumSubstrings() Method

```
int NumSubstrings();
```

The `NumSubstrings()` simply returns the number of substrings that were extracted during the most recent call to `Match()`.

2.1.4 The Dump() Method

```
void Dump();
```

The `Dump()` method dumps a list of all the substrings extracted during the most recent call to `Match()` to the terminal window. This method is intended to be used primarily during development. When constructing complex regular expressions, it is often difficult to know with certainty at which index in the substring list a particular substring will be placed. Using `Dump()` to display the substring list for a test case is a quick way to find out. Calls to `Dump()` don't typically appear in production code.

2.2 Private Methods

2.2.1 The CompilePattern() Method

```
void CompilePattern(std::string const &pattern);
```

`CompilePattern()` is a private method that serves as a helper to the class constructor. Only the constructor calls it. It is a wrapper around the PCRE library's `pcre_compile()` function. Its purpose is to convert the numeric error returned by `pcre_compile()` (if any) into a human-readable text message, that then gets thrown as a `std:string` exception.

2.2.2 The ConvertPercentString() Method

```
static std::string ConvertPercentString(std::string const &pattern);
```

`ConvertPercentString()` is a private method that serves as a helper to the class constructor. Only the constructor calls it.

The PCRE library expects regular expressions to be defined using standard Perl syntax. However, as mentioned previously, Perl syntax prefixes many character-class codes with backslash characters. Because backslashes are inconvenient to work with in C/C++ string literals, the `RE_c` class uses percent signs instead. The `ConvertPercentString()` method simply converts those percent signs to backslash characters to be used internally by the PCRE library.

2.3 Substring Access Operators

Although regular expressions can quickly recognize whether or not a given string matches a particular (sometimes extremely complex) pattern, their true power comes from the ability to recognize and extract certain user-defined substrings (parenthesized sub-patterns within the full regular expression) from the string under test. Two operators have been overloaded that allow the user to access these substrings (also see the `Dump()` method).

2.3.1 The [] Operator

```
const char* operator[](int ndx);
```

The `[]`, or “indexing,” operator takes an index into the substring list (the list of substrings extracted during the last call to `Match()`), and returns the corresponding substring.

Although internally, substrings are stored as STL strings, the `[]` operator returns substrings as a pointer to a `char` data type (i.e. a C-style string). Typically the STL string representation is preferred, but there are many instances in the code base in which the extracted substring is immediately passed to a system library function that is expecting a C-style string. The `[]` operator performs that conversion internally and really just serves as a convenience to the developer.

Index values that are out of range result in a pointer to a null string being returned. This is not a `NULL` pointer, but rather a pointer to a character array that contains the C-style string terminator (`\0`) as its only character.

2.3.2 The `()` Operator

```
std::string const operator()(int index);
```

The `()` operator takes an index into the substring list (the list of substrings extracted during the last call to `Match()`), and returns the corresponding substring. The returned substring is an STL string.

Index values that are out of range result in an empty STL string being returned.



Pacific Northwest
NATIONAL LABORATORY

*Proudly Operated by **Battelle** Since 1965*

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99352
1-888-375-PNNL (7665)
www.pnnl.gov



U.S. DEPARTMENT OF
ENERGY