# Henry Spencer's Regexp Engine Revisited

**Sunspot**

2 Jul 2003

A small, Unicode-aware regular expression engine based on Henry Spencer's early work

**Download source files - 35 Kb**

**Download demo project - 82 Kb**

## Introduction

Regular expressions (sometimes known as regexps) are used to describe text pattern matching in a concise form. They are useful whenever you need to apply such pattern matching: input validation, lightweight lexing, parsing email addresses, and so on. Many scripting languages (such as Perl and Python) have a built-in regular expression engines. One is also provided with the .NET framework.

For C++ programmers, there are several regular expression libraries available "in the wild" already. They are faster and more up to date than this one. They also understand more complex regular expressions (such as Perl regexps or POSIX regexps). One of those is `Boost.Regex`; it's even documented on CodeProject, and I highly recommend it in general.

## Why provide this one, then?

One answer: code footprint. This library is based on Henry Spencer's early public-domain regular expression implementation. When compiled with the test program under a Linux/ELF32 system, weights around 20 KB. Under Win32, you get a bit more than 19 KB in release mode. Most full POSIX-compliant regular expression engines (such as `Boost.RegExp`, PCRE, GNU Regex, or Henry Spencer's newest library) usually weight in around 50 KB with similar compilation options.

Code footprint may not seem to be a big deal in those days of 100 GB hard drives and multi megabyte applications. However, there may be reasons you need a small footprint - you may want to use regular expressions on a Pocket PC, for instance, or put it in a downloadable ActiveX control.

In addition, sometimes you don't need the full POSIX regular expressions; simple regexes will do find. Or you may want simple code so you can understand the implementation more easily. In those cases, this small engine is just the deal.

# Background

To use this library, you need to know the basic regular expression syntax. A good introduction can be found at this page. This particular implementation is a superset of the "extended regular expression" dialect.

Essentially, the following are available:

1. Metacharacters ".", "[...]" and "[^...]";
2. "+", "*" and "?" quantifiers;
3. Anchors "^", "$", "\<" and "\>";
4. Alternation "|";
5. Subexpressions "(...)";
6. Cheap character classes "*x*" where *x* is a letter which maps to a primitive ctype.h function, as follows:
    1. "\m": `isalnum()`
    2. "\a": `isalpha()`
    3. "\b": `isblank()` (GNU extension to ctype.h; essentially maps space and tab)
    4. "\c": `iscntrl()`
    5. "\d": `isdigit()`
    6. "\g": `isgraph()`
    7. "\l": `islower()`
    8. "\p": `isprint()`
    9. "\n": `ispunct()`
    10. "\s": `isspace()`
    11. "\u": `isupper()`
    12. "\x": `isxdigit()`
    13. "\w": `isword()` (not a *ctype.h* routine, matches `isalnum()` + the underscore character)

    Capitalize the letter to get the reverse match (for instance, "\D" matches every character for which `isdigit()` returns `false`)

In other words, it does *not* support:

1. Numbered matches "{n}" or "{n,m}" (however, those may be possible to add, at cost of relatively high memory usage).
2. Beginning or end of word matches "\b" and "\B" (those map to the `isblank()` character class instead); note that "\<" and "\>" can serve roughly the same purpose.
3. Backreferences within the regular expression (those were singlehandedly responsible for forcing Mr. Spencer's rewrite of his library, as far as I understand).
4. POSIX collating elements "[.*char*.]" and "[=*char*=]".
5. POSIX character classes "[:*class*:]" (however, see the Cheap character classes described above).
6. Case insensitive matches and "basic" regular expressions.
7. Perl-like character classes.

In general, consider the engine to support the extended, pre-POSIX regular expression syntax, without the Perl extensions, and with character classes added "on top" by me as a somewhat ugly hack.

As mentioned before, the original regexp engine was written by Henry Spencer; I found it at ftp://ftp.zoo.toronto.edu/ and modified to support wide characters with the appropriate preprocessor definition. I also extended it to work with an arbitrary number of subexpressions (the original code was limited to 9). The resulting interface is close to the POSIX regex interface, but not quite the same, unfortunately. This gives it a re-entrant interface (the original interface definitely was not re-entrant). I basically tailored the interface so it would be easy to make it work with dynamic memory allocation.

# Using the code

## The C interface

You have three ways of using that code. The basic C interface (found in the <file>regexp.h header) will work fine. It is used like this:

```
#include "regexp.h"
#include <string>
#include <vector>

int parse_email(const std::string& to_match,
                std::string& user_name,
                std::string& host_and_domain,
                std::string& domain_suffix)
{
    regexp* compiled;                                           // line A

    int retval = re_comp(&compiled,
                         "^([A-Za-z0-9]+)@(.+)\\.(\\a+)$");     // line B
    if(retval < 0)
        return retval;                                          // line C

    regmatch* matches = new regmatch[re_nsubexp(compiled)];     // line D

    retval = re_exec(compiled,
                     to_match.c_str(),
                     re_nsubexp(compiled),
                     &matches[0]);                              // line E
    re_free(compiled);                                          // line F

    if(retval < 1)                                              // line G
    {
        delete[] matches;
        return retval;
    }

    user_name = std::string(to_match.begin() + matches[1].begin,
                            to_match.begin() + matches[1].end); // line H

    host_and_domain = std::string(to_match.begin() + matches[2].begin,
                                  to_match.begin() + matches[2].end);
    domain_suffix = std::string(to_match.begin() + matches[3].begin,
                                to_match.begin() + matches[3].end);

    delete[] matches;
    return 1;
}

int main(int argc, char* argv[])
{
    if(argc >= 2)
    {
        std::string user_name, host_and_domain, domain_suffix;
        if(parse_email(argv[1], user_name,
            host_and_domain, domain_suffix) < 1)
        {
            printf("Not an email address\n");
            return 1;
        }
        printf("User name: %s\nHost/domain: %s\nDomain suffix: %s\n",
               user_name.c_str(),
```

```
                host_and_domain.c_str(),
                domain_suffix.c_str());
        return 0;
    }

    printf("Usage: %s <email address>\n", argv[0]);
    return 1;
}
```

The program above contains a cheap email parser (the regular expression is not compliant with any RFC, it just seems to work for the two test addresses I gave it). It splits the email in user name, domain prefix, and domain suffix (the domain suffix is, say, the ".com" at the end of an address).

Here are explanations of the interesting lines in function `parse_email()`:

1. LINE A: regular expressions must be compiled before they can be used. Compiled regular expressions are returned through a pointer to an opaque type, `regexp`. You must therefore declare a `regexp*` to hold the compiled regular expression.
2. LINE B: this statement creates a compiled regular expression. The resulting compiled expression is placed in `&compiled`; the second argument is the regular expression itself. Note that since we express it as a constant C string, we need to double every backslash in the expression.

   The expression works thus: first match any alphanumeric characters, up until the @ sign; then, match one or more of any characters, followed by a "." (matching is "greedy", which means the last "." of the expression will terminate the second subexpression); finally, match one or more alphabetic characters.
3. LINE C: `re_comp()` returns 0 on success, or a negative error code on failure. Error codes are defined in <file>regexp.h and can be any of:
   1. `REGEXP_BADARG`: invalid argument
   2. `REGEXP_ESIZE`: regular expression too big
   3. `REGEXP_ESPACE`: out of memory
   4. `REGEXP_EPAREN`: parenteses () not balanced
   5. `REGEXP_ERANGE`: invalid character range
   6. `REGEXP_EBRACK`: brackets [] not balanced
   7. `REGEXP_BADRPT`: quantifier operator invalid
   8. `REGEXP_EESCAPE`: invalid escape \ sequence
   9. `REGEXP_EEND`: internal error!
4. LINE D: To attempt to match a compiled regular expression, use the `re_exec()` function. That function takes an array of `regmatch` structs. There should be one element in the array per subexpression in the regexp. In line D, we allocate an array to store the matches. The `re_nsubexp()` function queries the compiled regular expression for the number of submatches it contains.
5. LINE E: Here, we finally try to match the regular expression. The first argument to `re_exec()` is the compiled regexp; then, pass the string to match, the number of elements in the match array, and the address to the match array. If you're not interested in the submatches, you can pass 0 as the third parameter and `NULL` as the last parameter. Make sure you have at least as many elements in the array you pass as the size passed in the third parameter.
6. LINE F: Once you're done with a compiled pattern, you should free it using `re_free()`. Please don't use `free()` or `delete[]` to free it (however, see the Customization section if you want to override the memory allocator used by the regular expression engine).

   Note that you should probably keep those compiled regular expressions around for the duration of the program. Since compilation is a relatively slow process, it's best to keep the compiled expression as long as it's possible you'll need it again.
7. LINE G: The `re_exec()` function can return a negative error code (the same as `@code{re_comp()}, 0` (meaning the string didn't match the regexp), or 1 (meaning the string did match the regexp).
8. LINE H: The `regmatch` structure contains two fields: `begin` and `end`. They contain offsets into the string passed to `re_exec()`. `begin` is the start offset of the submatch, and `end` is the offset of the character one-past-the-end of the

submatch (hence, `end - begin` is the length of the submatch).

Element 0 of match array is always the match of the full regexp. Element 1 is the first subexpression (counting from the left), and so on.

If a given element was not matched, its `begin` and `end` fields will be set to -1.

In the example statement, we use the offsets to calculate the begin and end iterator of the substring for the first subexpression.

Note that if you compile the regexp library with the `REGEXP_UNICODE` preprocessor symbol defined, you get wide character versions of the `re_comp()` and `re_exec()` routines. Those are accessed through the `re_comp_w()` and `re_exec_w()` routines. They work exactly like their non-"`_w`" versions, except that they take wide character strings instead of multibyte strings. Here's the example function rewritten for wide characters:

```
#define REGEXP_UNICODE
#include "regexp.h"
#include <string>
#include <vector>

int parse_email(const std::wstring& to_match,
                std::wstring& user_name,
                std::wstring& host_and_domain,
                std::wstring& domain_suffix)
{
    regexp* compiled;                                           // line A

    int retval = re_comp_w(&compiled,
                           L"^([A-Za-z0-9]+)@(.+)\\.(\\a+)$");  // line B
    if(retval < 0)
        return retval;                                          // line C

    regmatch* matches = new regmatch[re_nsubexp(compiled)];     // line D

    retval = re_exec_w(compiled,
                       to_match.c_str(),
                       re_nsubexp(compiled),
                       &matches[0]);                            // line E
    re_free(compiled);                                          // line F

    if(retval < 1)                                              // line G
    {
        delete[] matches;
        return retval;
    }

    user_name = std::wstring(to_match.begin() + matches[1].begin,
                             to_match.begin() + matches[1].end); // line H

    host_and_domain = std::wstring(to_match.begin() + matches[2].begin,
                                   to_match.begin() + matches[2].end);
    domain_suffix = std::wstring(to_match.begin() + matches[3].begin,
                                 to_match.begin() + matches[3].end);

    delete[] matches;
    return 1;
}
```

## The Class Interface

As an example, and also for convenience, the demo code contains a `CRegExp` class which can serve as a nice interface to the library. That interface uses the `CString` class which is available in WTL (note that it can probably work with MFC as well - I just haven't tested it).

Here's the class declaration for the `CRegExp` class (private details omitted):

```
class CRegExpException
{
public:
    CRegExpException(int nError);

    int GetError() const;
    CString GetErrorString() const;
};

class CRegExp
{
public:
    CRegExp(LPCTSTR pszPattern);
    ~CRegExp();

    BOOL Exec(const CString& pszMatch);
    BOOL IsMatched(int nSubExp = 0) const;
    int GetMatchStart(int nSubExp = 0) const;
    int GetMatchEnd(int nSubExp = 0) const;
    CString GetMatch(int nSubExp = 0) const;
    int GetNumberOfMatches() const;
};
```

An instance of the `CRegExpException` class is thrown whenever the library returns an error; `GetError()` returns the error code (as described in the C interface section), and `GetErrorString()` returns a plain English string associated to the error.

The `CRegExp` class has the following members:

1. `CRegExp::CRegExp(LPCTSTR pszPattern)`
   Creates a new `CRegExp` object with regular expression `pszPattern`.
2. `int CRegExp::GetNumberOfMatches()`
   Returns the number of matches that can be expected from the pattern which was passed as construction, which is equal to the number of subexpressions in the pattern, *plus one*. For instance, if the pattern contains two subexpressions, `GetNumberOfMatches()` returns 3.
3. `BOOL CRegExp::Exec(const CString& sMatch)`
   Attempts to match `sMatch` with the regular expression specified at construction time. Returns `TRUE` if the expression was matched, `FALSE` otherwise.
4. `int CRegExp::GetMatchStart(int nSubExp = 0)`
   `int CRegExp::GetMatchEnd(int nSubExp = 0)`
   `CString CRegExp::GetMatch(int nSubExp = 0)`
   `BOOL CRegExp::IsMatched(int nSubExp = 0)`
   Can only be called after a call to `CRegExp::Exec()`. `GetMatchStart()` returns the starting offset of the `nSubExp`'th subexpression (with 0 meaning the whole expression, 1 the first subexpression, and so on). `GetMatchEnd()` returns the offset of one-past-the-end of the `nSubExp`'th subexpression. `GetMatch()` returns the string which matched the `nSubExp`'th subexpression.

   `IsMatched()` simply returns whether the `nSubExp`'th subexpression was matched.

   Note that when you specify a subexpression that was either not matched or one which is past the range of valid subexpressions, `GetMatchStart()` and `GetMatchEnd()` return -1, and `GetMatch()` returns an empty string.

Here is the C example (from the C interface section) reworked for wrapper use:

```
BOOL ParseEMail(const CString& sToMatch,
                CString& sUserName,
                CString& sHostAndDomain,
```

```
                CString& sDomainSuffix)
{
    CRegExp reEMailExpr(_T("^([A-Za-z0-9]+)@(.+)\\.(\\a+)$"));
    if(reEMailExpr.Exec(sToMatch) == FALSE)
        return FALSE;

    // the regular expression's format should ensure that all
    // three expressions match, or the expression doesn't match
    // at all.
    ATLASSERT(reEMailExpr.IsMatched(1) &&
              reEMailExpr.IsMatched(2) &&
              reEMailExpr.IsMatched(3));

    sUserName = reEMailExpr.GetMatch(1);
    sHostAndDomain = reEMailExpr.GetMatch(2);
    sDomainSuffix = reEMailExpr.GetMatch(3);

    return TRUE;
}
```

## Standard C++ Wrapper

For those of you who do not have access to `CString` but do have access to the Standard C++ Library, you have not been forgotten. The wrapper provided here is not in the same class as `Boost.RegExp` (which is *much* more comprehensive and is optimized for minimal heap usage), but it should be sufficient to make the C interface more friendly.

Here's the class declaration for the `regular_expression` class (private details omitted):

```
class regular_expression_error : public std::runtime_error
{
public:
    regular_expression_error(int error_code, regexp* re);

    int code() const;
    const char* message() const;
};

class regular_expression
{
public:
#ifdef REGEXP_UNICODE
    typedef wchar_t CharT;
    typedef std::wstring string_type;
#else
    typedef char CharT;
    typedef std::string string_type;
#endif
    typedef typename string_type::size_type size_type;
    typedef typename string_type::const_iterator const_iterator;

    regular_expression(const CharT* pattern);
    regular_expression(const string_type& pattern);

    bool exec(const CharT* match);
    bool exec(const string_type& match);

    bool matched(size_type sub_exp = 0) const;
    const_iterator begin(size_type sub_exp = 0) const;
    const_iterator end(size_type sub_exp = 0) const;
    string_type operator[](size_type sub_exp) const;
    size_type size() const;
};
```

An instance of the `regular_expression_exception` class is thrown whenever the library returns an error; `code()` returns the error code (as described in the C interface section), and `message()` returns a plain English string associated to the error.

The `regular_expression` template has the following members:

1. `regular_expression(const CharT* pattern)`
   `regular_expression(const string_type& pattern)`
   Creates a new `regular_expression` object with regular expression passed in `pattern`. The regular expression is compiled, and the compiled representation remains until the `regular_expression` instance is destroyed.

2. `size_type size()`
   Returns the number of matches that can be expected from the pattern passed to the constructor. This is equal to the number of subexpressions in the pattern, *plus one* (the 0'th match, which is the whole expression, is always present).

3. `bool exec(const CharT* match)`
   `bool exec(const string_type& match)`
   Attempts to match `match` with the regular expression compiled in the object. Returns `true` if the expression was matched, and `false` otherwise.

   Note that `match` is copied within the instance of the class, to ensure that `begin()`, `end()` and `operator[]` work properly regardless of what happens to `match`.

4. `const_iterator begin(size_type sub_exp = 0)`
   `const_iterator end(size_type sub_exp = 0)`
   `bool matched(size_type sub_exp = 0)`
   `string_type operator[] (size_type sub_exp)`
   Those methods can only be called after a successful call to `exec()`.

   `begin()` returns an iterator pointing to the start of the `sub_exp`'th subexpression (with 0 being the whole expression, 1 being the first subexpression, and so on).

   `end()` works the same way, except that it returns an iterator point to one-past-the-end of the `sub_exp`'th subexpression.

   `matched()` returns whether the `sub_exp`'th subexpression was matched.

   Finally, `operator[]` is equivalent to `string_type(begin(sub_exp), end(sub_exp))`, that is, it returns the `sub_exp`'th match.

   Note that when passing an invalid `sub_exp` (either because that subexpression was not matched, or because `sub_exp` is greater than `size()`), the range represented by `begin()` and `end()` will be empty, and `operator[]` will return an empty `string_type`.

Note that unlike a "traditional" Standard Library class, this is not a template; rather, it's a class that has two personalities depending on the definition (or lack thereof) of preprocessor symbol `REGEXP_UNICODE`. Given that the underlying C code is not templatized with the character type, I had little choice in the matter.

Here is the C example (which should look *very* familiar by now, otherwise see the C interface section) reworked in all its standard library glory:

```
bool parse_email(const std::string& to_match,
                 std::string& user_name,
                 std::string& host_and_domain,
                 std::string& domain_suffix)
{
    regular_expression email_expr("^([A-Za-z0-9]+)@(.+)\\.(\\a+)$");
    if(!email_expr.exec(to_match))
        return false;

    // the regular expression's format should ensure that all three
    // expressions match, or the expression doesn't match at all.
    assert(email_expr.matched(1) &&
```

```
            email_expr.matched(2) &&
            email_expr.matched(3));

    user_name = email_expr[1];
    host_and_domain = email_expr[2];
    domain_suffix = email_expr[3];

    return true;
}
```

# How to include the library in your project

I haven't provided a project file for the library itself because you probably won't want to make a DLL out of this library. Its code footprint is small enough to link it statically.

The following files serve as the "core" of the library:

- *regexp.h* (public header)
- *regexp_int.h* (internal header)
- *regexp_custom.h* (internal header)
- *regmagic.h* (internal header)
- *memory.c* (memory allocation routines)
- *regerror.c* (implementation of error translation function)
- *regexp.c* (main implementation file)
- *report.c* (default report functions)
- *widechar.c* (default implementation for Unicode character support)

The following files are optional and are only needed if you want to use the "old-style" interface and the substitution functions:

- *frontend.c*
- *regsub.c*

Finally, the following files are unit tests inherited from the original source files; you probably don't need them, but they are provided together with the rest of the library for consistency purposes:

- *try.c*
- *timer.c*
- tests

Depending on which wrapper you may want to use, you can also add the *wtl/CRegExp.** or the *stl/stdregexp.** files to your project. The demo projects show how this can be done.

# Customization

Libraries such as this one tend to be used in a variety of context. Hence, I've tried to isolate the dependencies on runtime library routines so they can be easily overridden.

By default, the library allocates memory using `malloc()` and `free()`. However, if you want to use a custom allocator, simply provide implementations of the following two routines:

```
extern "C" void* re_malloc(size_t sz);
extern "C" void re_cfree(void* p);
```

Those should have similar semantics to `malloc()` and `free()`. The provided memory.c file contains a default implementation which uses standard `malloc()` and `free()`; if you want to override this implementation, simply provide your own and don't link with the provided memory.c file.

In addition, when compiling with `REGEXP_UNICODE`, there is some attempt to providing a multibyte interface for those who work with mixed wide character/multibyte strings. It doesn't work quite well (subexpression offsets are not calculated properly), but if you are interested in using it, you'll definitely want to override the following two functions:

```
extern "C" wchar_t* re_ansi_to_unicode(const char* s);
extern "C" char* re_unicode_to_ansi(const wchar_t* s);
```

Those functions should use `re_malloc()` to create a new string of the appropriate type. By default, the C library `mbstowcs()` and `wcstombs()` routines are used; however, you may want to map those to, say, `::MultiByteToWideChar()`and `::WideCharToMultiByte()`. The default implementation is in *widechar.c*; provide your own implementation and don't link with *widechar.c* to override the default behaviour. Note that you'll also have to provide an implementation of `isblank()` and `iswblank()` if you don't use the provided *widechar.c*; see comments in the file for details.

Finally, the regular expression library calls a function to report internal errors in a more fine-grained manner than through the `REGEXP_*` error codes. By default, reporting is done through `fprintf(stderr, ...)`. To override this behaviour, provide

```
extern "C" void re_report(const char* error);
```

(Note that the error is always provided in plain chars) You could provide, for instance,

```
extern "C" void re_report(const char* error)
{
    char buffer[128];
    ::wsprintfA(buffer, "REGEXP ERROR: %s\n", error);
    ::OutputDebugStringA(buffer);
}
```

The default implementation is in file *report.c*. Simply provide your own implementation and don't link with *report.c* if you don't like the default implementation.

# The Demo Projects

The first demo program (*re2demo.exe*) is a simple WTL dialog application which allows you to explore different regular expressions. The top field should contain the regular expression; the middle field should contain the string to match. Once you press the "Try It" button, the matches will be placed in the bottom combo box (open the combo box to see all the submatches). If there was an error, or the string didn't match the regexp, the error will be printed as the first (and only) entry of the combo box.

Note that the demo program is not meant to be a demonstration of clean WTL style. It's mostly an example of how to integrate the regular expression engine in your own programs.

*(Actually, I really should confess that I only wrote it to make sure the CRegExp class works properly)*

The second demo program (*try.exe*) is a simple unit testing program which was provided in the original source code archive. It is provided in pre-compiled form simply as a convenience.

# Future Directions

A few additional utilities, such as a string substitution routine and a global match routine, could be added with relatively little trouble. I've not done this yet in the interest of posting this code quickly.

In addition, I'm pretty sure it would be possible to support the `{n}` and `{m,n}` quantifiers (although it may be costly in terms of memory usage). I may decide to add this eventually.

Also, the fact that the plain char versions don't work right when `REGEXP_UNICODE` is defined is definitely a bug. The workaround right now is to have two separate libraries--one Unicode, one multibyte. It's a hack, and I dislike keeping such things around.

Finally, I seem to recall some modified version of this regexp engine which optimized some common case to yield better performance. I may eventually hunt it down and apply those modifications to the version provided here.

# Related Work

There is another CRegExp class which exists in the wild. The author of this class is to thank for the inspiration of my implementation; that class made me aware of the availability of Mr. Spencer's code. However, one thing I didn't do was merge the C routines inside my own `CRegExp` class, the way it's done in the other implementation. Mainly, I wanted the code to remain easily customizable for non-MFC people, yet still benefit from the modifications I did (Unicode support and so on). The easiest way to do this is to keep the plain C interface around.

Before I ended up with this specific implementation, I tried to extract Mr. Spencer's latest implementation (which is buried somewhere in the Tcl/Tk code). I managed to extract it, but was disappointed by the rather large code footprint.

I also considered PCRE; unfortunately, their Unicode support was still experimental at the time, and it's based on UTF-8 encoded strings rather than UCS-2 wide character strings and I needed UCS-2. It's unfortunate because it is relatively small and it's supposed to be a very fast library. Oh, well.

Finally, there are articles on CodeProject about the same subject. They provide tutorials for different libraries. You may want to look at those for alternative solutions.

# Conclusion

We have seen a short tutorial on how to use the regular expression package provided with this article. Also, we've seen how to use the C++ two classes provided as example wrappers around the package.

From the comments on the regular expression syntax, it should be clear by now that this is not the most complete, nor the fastest, library available. However, it's simple, easy to understand, portable, and small. If you're looking for any of those criteria over completeness and speed, this library will fit your needs better.

I hope you'll enjoy using this as much as I enjoyed tweaking its code.

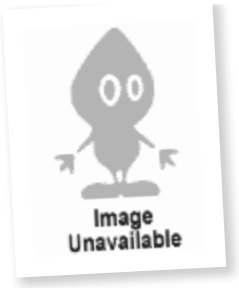
# History

- 2003/06/28 - Initial revision
- 2003/07/03 - Added missing file to demo project

# License



# About the Author




**Sunspot**
Web Developer
Canada

I'm a senior software developer, working at Silanis Technology (http://www.silanis.com). I've acquired quite a bit of experience (usually the hard way!) in Win32 and raw COM programming on the job. In my spare time, I like to monkey around with POSIX code.

I'm mostly interested in portable C++ libraries. I'm happiest when I develop portable C++ code--C++ being such a powerful language as long as one keeps clear of the rather nasty subtleties of the language.

I hope the articles I contribute will be of some help to someone. If even one person gains a few hours through use of that code, I'll be very happy.

When not coding, I like to listen to Anime and try to learn Japanese. It's not working too well so far, unfortunately. :{)

# Comments and Discussions

**41 messages** have been posted for this article Visit **https://www.codeproject.com/Articles/4421/Henry-Spencer-s-Regexp-Engine-Revisited** to post and view comments on this article, or click **here** to get a print view with messages.