Regex In C++11 And Boost

Perl! Templates! Iterators!

By Your Powers Combined, I Am CAPTAIN REGEX!

Stephan T. Lavavej
("Steh-fin Lah-wah-wade")
Visual C++ Libraries Developer
stl@microsoft.com

regex: Flexible String Processing

- Validation: Is this input well-formed?
  - Example: Is this a serial number?
- Decision: What set does this string belong to?
  - Example: Is this filename a JPEG? Is it a PNG?
- Parsing: What is in this input?
  - Example: What is the year of this date?
- Transformation: Format strings for output or further processing
  - Example: Escape special characters
- Iteration: Find each occurrence of a pattern within a string
  - Example: Iterate through all URLs within a string
- Tokenization: Systematically split apart a string
  - Example: Break a string into whitespace-separated words

regex: Robust String Processing

- Processing strings with regex is superior to handwritten code
  - Control flow is difficult to understand and modify
  - regex simplifies control flow, moving the description of string processing into the regular expression
  - Regular expressions are closer to the problem domain than code, abstracting away much code complexity
  - Even intricate regular expressions are easier to understand and modify than equivalent code
- regex uses STL techniques to achieve both generality and simplicity

Regular Expression Refresher: Overview

- A regular expression is a pattern
  - Represented by a string
  - Extremely compact, potentially inscrutable
- This pattern is applied to a target string
  - Match: Does the pattern describe the entire target?
  - Search: Does the pattern describe part of the target?
  - Replace: Transform substrings (described by the pattern) of the target
    - The transformation is done according to another pattern
    - Represented by a format string
    - Format grammar is simpler than regular expression grammar
- Applying a regular expression is also called "matching"

What Grammars Can `regex` Use?

- `ECMAScript` – JavaScript's standard, based on Perl
  - The default, and the most powerful
  - Supports more features than any other grammar
  - Lacks no features (except awk's octal escapes)
- `basic` – POSIX Basic Regular Expressions
- `extended` – POSIX Extended Regular Expressions
- `awk` – POSIX `awk` utility
- `grep` – POSIX `grep` utility
- `egrep` – POSIX `grep -E` utility

Regular Expression Refresher: Capture Groups

- A regular expression can contain capture groups
- Capture groups…
  - … are sometimes called subexpressions
  - … identify specific parts of the regular expression for later reference
    - While matching: Backreferences
    - After matching: Drill down into match results, asking which capture groups matched where
    - While replacing a substring S with a replacement R
      - R can be a fixed string, or…
      - R can be built from parts of S matched by capture groups
      - Those parts of S can be reordered and duplicated

Regular Expression Refresher: Precedence

- `ab+c|d` matches `abc`, `abbc`, `abbbc`, …, and `d`
  - Elements (for example: `b`) are the building blocks
  - Quantifiers (for example: `b+`) bind most tightly
  - Concatenation (for example: `ab+c`) binds next
  - Alternation (for example: `ab+c|d`) binds most weakly
- Parentheses create elements for grouping
  - `a|b+` matches `a`, `b`, `bb`, `bbb`, etc.
  - `(a|b)+` matches `a`, `b`, `aa`, `ab`, `ba`, `bb`, `aaa`, etc.

Regular Expression Refresher: Elements (Ordinary, Wildcard, Anchor)

- An ordinary character matches itself
  - Case sensitive matching by default
  - Special characters: `. \ * + ? | ^ $ ( ) [ ] { }`
- A wildcard `.` matches any single character except newline
- The anchors `^` and `$` match empty substrings at the beginning and end of the target string
  - `^cat` matches a substring of `catch` but not of `kittycat`
  - `cat$` matches a substring of `kittycat` but not of `catch`

Regular Expression Refresher: Elements (Bracket Expressions)

- A bracket expression matches any single character in a specific set, or *not* in a specific set
  - `[ch]at` matches `cat` and `hat`
  - `[2-4]7` matches `27`, `37`, and `47`
  - `[^b]at` matches `aat`, `cat`, `3at`, etc. but not `bat`
- Bracket expressions can contain character classes
  - `[[:xdigit:]]` matches any hexadecimal digit
  - `[^[:xdigit:]]` matches any non-hexit

Regular Expression Refresher: Elements (Escapes)

- `\` does many things:
  - Special character escapes: `\.` `\\` `\*` `\+` etc.
  - Backreferences: `\1` `\2` `\3` etc.
  - File format escapes: `\f` `\n` `\r` `\t` `\v`
  - DSW escapes: `\d` `\s` `\w` `\D` `\S` `\W`
    - `\d` matches digit characters
    - `\s` matches whitespace characters
    - `\w` matches word characters (alphanumeric and underscore)
    - `\D` matches non-digit characters, etc.
  - Word boundaries: `\b` `\B`
    - `\b` matches the empty substrings at the beginning and end of a word, `\B` is the opposite
  - Hex escapes, Unicode escapes, control escapes

We Put A Backslash In Your Backslash So You Can Escape While You Escape

- Backslashes are also special to C++
- String literals must contain double backslashes in order to present single backslashes to `regex`
- To match a filename: `"meow\\.txt"`
- And `"directory\\\\meow\\.txt"`
- C++11 raw string literals solve this problem:
  - `R"(C:\\Temp\\meow\.txt)"`

Regular Expression Refresher: Elements (Groups, Asserts)

- `(whatever)` overrides precedence and creates a capture group
- `(?:whatever)` overrides precedence only (hence, "noncapture group")
  - Usually, creating a capture group is okay even if you're not interested in it
- For people who want to be really clever:
  - Positive asserts: `(?=whatever)`
  - Negative asserts: `(?!whatever)`

Regular Expression Refresher: Quantifiers

- `*` means "0 or more"; `ab*c` matches `ac`, `abc`, `abbc`, etc.
- `+` means "1 or more"; `ab+c` matches `abc`, `abbc`, etc.
- `?` means "0 or 1"; `ab?c` matches only `ac` and `abc`
- `{3}` means "exactly 3"
- `{3,}` means "3 or more"
- `{3,5}` means "3 to 5 inclusive"
- `*` is an abbreviation for `{0,}`
- `+` is an abbreviation for `{1,}`
- `?` is an abbreviation for `{0,1}`

Regular Expression Refresher: Non-Greedy Quantifiers

- "Greed is good" – Gordon Gekko
  - Quantifiers are greedy by default, matching as many characters as they can
  - When simply matching, this doesn't matter
- … except when it's bad
  - When matching and examining submatches, or when searching, or when replacing, you may want quantifiers to match as few characters as they can
- Append `?` for non-greedy matching
  - `<b>.*</b>` matches `<b>x</b>y<b>z</b>`
  - `<b>.*?</b>` matches `<b>x</b>y<b>z</b>`

What Does regex Work With?

- As with STL algorithms, iterators are used to decouple `regex` from the data that it manipulates
  - `const char *` and `string::const_iterator`
  - `const wchar_t *` and `wstring::const_iterator`
- The most general overloads take `[first, last)` for maximum generality
- Convenience overloads are provided for:
  - `std::string`
  - `std::wstring`
  - Null-terminated `const char *`
  - Null-terminated `const wchar_t *`

regex Types

- `basic_regex`: A finite state machine constructed from a regular expression pattern
  - More than meets the eye: A complex data structure that looks like it stores a plain old string
- `match_results`: A representation of a substring that matches a regular expression, including which capture groups match where
- `sub_match`: An iterator pair representing a substring that matches an individual capture group

regex Algorithms

- `regex_match()` and `regex_search()`
  - Match: Does a pattern describe a string in its entirety?
    - If so, which capture groups matched where?
  - Search: Does a pattern describe some part of a string?
    - If so, where is the first substring described by the pattern?
    - And, which capture groups matched where?
- `regex_replace()`
  - Replace: Transform all occurrences of a pattern in a string according to a given format
    - Optional: Transform just the first occurrence
    - Optional: Remove the non-transformed parts of the string

regex Iterators

- `regex_iterator`
  - Iterate through all occurrences of a pattern in a string
  - `list<T>::iterator` – node traversal in iterator form
  - `regex_iterator` – `regex_search()` in iterator form
- `regex_token_iterator`
  - Iterate through the capture groups of all occurrences of a pattern in a string
    - Filter down to one capture group of interest
    - Filter down to several capture groups of interest
    - Field splitting: iterate through what *doesn't* match
  - Extremely powerful for parsing

regex Typedefs: Because Typing string::const_iterator Isn't Fun

Typedef	True Name
string	basic_string<char>
regex	basic_regex<char>
cmatch	match_results<const char *>
smatch	match_results<string::const_iterator>
csub_match	sub_match<const char *>
ssub_match	sub_match<string::const_iterator>
cregex_iterator	regex_iterator<const char *>
sregex_iterator	regex_iterator<string::const_iterator>
cregex_token_iterator	regex_token_iterator<const char *>
sregex_token_iterator	regex_token_iterator<string::const_iterator>

regex_match(): Simple Matching

```
const regex r("[1-9]\\d*x[1-9]\\d*");
for (string s; getline(cin, s); ) {
    cout << (regex_match(s, r) ? "Yes" : "No") << endl;
}
```

- Prints:

```
2x4
Yes
2560x1600
Yes
007x006
No
a5x5b
No
```

regex's Constructor Is explicit

- Writing this:

```
regex_match(s, "[1-9]\\d*x[1-9]\\d*")
```

- Triggers 6 compiler errors, starting with:

```
error C2784: 'bool std::regex_match(
  const std::basic_string<_Elem,_StTraits,_StAlloc> &,
  const std::basic_regex<_Elem,_RxTraits> &,
  std::regex_constants::match_flag_type)'
  : could not deduce template argument for
  'const std::basic_regex<_Elem,_RxTraits> &' from
  'const char [18]'
```

- regex's constructor is explicit because it can be expensive

regex_match(): Using match_results

```
const regex r("([1-9]\\d*)x([1-9]\\d*)");
for (string s; getline(cin, s); ) {
    smatch m;
    if (regex_match(s, m, r)) {
        cout << m[1] << " by " << m[2] << " is "
        << stoi(m[1]) * stoi(m[2]) << " pixels" << endl;
    }
}
```

- Prints:

```
2560x1600
2560 by 1600 is 4096000 pixels
```

regex_search(): A Variant Of regex_match()

```
const regex r("//");
for (string s; getline(cin, s); ) {
  smatch m;
  if (regex_search(s, m, r)) {
    cout << "Comment: [" << m.suffix() << "]" << endl;
  }
}
```

- Prints:

```
Nothing here.
++i; // Silly comment.
Comment: [ Silly comment.]
--i; // Nested // comment.
Comment: [ Nested // comment.]
```


Format String Refresher

- Example:
  - Regex: `([A-Z]+)-([0-9]+)`
  - String: `2161-NCC-1701-D`
- Escape sequence – Replaced by:
  - `$1` – What matches the 1st capture group (e.g. `NCC`)
  - `$2` – What matches the 2nd capture group (e.g. `1701`)
  - `$&` – What matches the whole regex (e.g. `NCC-1701`)
  - `$\`` – What appears before the whole regex (e.g. `2161-`)
  - `$'` – What appears after the whole regex (e.g. `-D`)
  - `$$` – `$`

regex_replace()

```
const regex r("(\\w+)( \\w+\\.?)? (\\w+)");
for (string s; getline(cin, s); ) {
    cout << "==> "
        << regex_replace(s, r, "$3, $1$2") << endl;
}
```

- Prints:

```
Stephan T. Lavavej
==> Lavavej, Stephan T.
Stephan Thomas Lavavej
==> Lavavej, Stephan Thomas
Stephan Lavavej
==> Lavavej, Stephan
```


sub_match

- Abbreviated class definition (omitting some contents):

```
template <typename BidiIt> class sub_match
    : public pair<BidiIt, BidiIt> {
public:
    typedef typename iterator_traits<BidiIt>::value_type value_type;
    typedef typename iterator_traits<BidiIt>::difference_type difference_type;
    bool matched;
    difference_type length() const;
    operator basic_string<value_type>() const;
    basic_string<value_type> str() const;
};
```

- csub_match and ssub_match convert to std::string

match_results: A Container Of sub_matches

- Highly abbreviated class definition:

```
template <typename BidiIt> class match_results {
public:
    size_t size() const;
    bool empty() const;
    const sub_match<BidiIt>& operator[](size_t n) const;
    const sub_match<BidiIt>& prefix() const;
    const sub_match<BidiIt>& suffix() const;
    string_type format(const string_type& fmt,
        regex_constants::match_flag_type flags =
        regex_constants::format_default) const;
};
```

- You just inspect `match_results`; only `regex_match()` and `regex_search()` can modify `match_results`

match_results Member Functions

- If `regex_match/search()` returns `false`:
  - `m.empty() == true` and `m.size() == 0`
  - DO NOT INSPECT any other part of `m`
- Otherwise, `m.empty() == false` and:
  - `m.size()` is 1 + the # of capture groups in the regex
  - `m[0]` is the entire match
  - `m[1]` is the 1st `sub_match`, `m[2]` is the 2nd, etc.
  - `m.prefix()` precedes the match, `m.suffix()` follows
  - `m.format(fmt)` acts like `regex_replace()`

Pitfall: Stepping Through A String With regex_search()

- NEVER use regex_search() to find successive occurrences of a regex in a string
- Ch. 19 of Pete Becker's TR1 book lists the problems:
  - Lost Anchors
  - Lost Word Boundaries
  - Empty Matches
- ALWAYS use regex_iterator instead
  - Robust: Correctly handles all regexes
  - Simple: Even easier than naively using regex_search()
  - Efficient: No additional overhead

regex_iterator: Iterate Through match_results (1/2)

```
const regex r("\\w*day");
string s;
getline(cin, s);
for (sregex_iterator i(s.begin(), s.end(), r), end;
    i != end; ++i) {
    cout << (*i)[0] << endl;
}
```

- regex_iterator's default ctor creates an end-of-sequence iterator
  - Like istream_iterator
  - Unlike vector<T>::iterator
- ALWAYS use a named regex, NEVER a temporary

regex_iterator: Iterate Through match_results (2/2)

- Prints:

```
Hate Mondays, love Tuesday; every day should be Caturday
Monday
Tuesday
day
Caturday
```

- `sregex_iterator::operator*()` returns a `const smatch&`
  - The `sregex_iterator` contains the `smatch`
  - Copy the `smatch` if you need to inspect it after incrementing the `sregex_iterator` (unusual)
- `sregex_iterator::operator->()` also works

regex_token_iterator: Iterate Through sub_match

- Just like `regex_iterator`, except for:
  - Different constructor arguments
  - `sregex_token_iterator::operator*()` returns `const ssub_match&` (`operator->()` is also different)
- You pick capture groups of interest (one or many)
  - Use them to construct a `regex_token_iterator`
  - They will be cyclically presented to you
- `regex_token_iterator` adapts `regex_iterator`
  - An iterator adaptor adaptor!

regex_token_iterator: Constructors

- Five ways to specify capture groups:

```
regex_token_iterator(BidiIt a, BidiIt b,
    const regex_type& r, XYZ,
    regex_constants::match_flag_type m =
    regex_constants::match_default);
```

- Where XYZ is one of:
  - `int submatch = 0`
  - `const vector<int>& submatches`
  - `initializer_list<int> submatches`
  - `const int (&submatches)[N]`

regex_token_iterator: 0^{th} Capture Group

- Rewriting the `regex_iterator` example:

```
const regex r("\\w*day");
string s;
getline(cin, s);
for (sregex_token_iterator i(s.begin(), s.end(), r), end;
    i != end; ++i) {
    cout << *i << endl;
}
```

- `*i` instead of `(*i)[0]`

regex_token_iterator: Field Splitting (1/2)

- Triggered by asking for capture group `-1`
- Iterates through what doesn't match the `regex`
- Infinitely better than `strtok()`, which is dangerous, limited, and inconvenient
- If the string ends with a field splitter:
  - Every token ends with a field splitter
- If the string doesn't end with a field splitter:
  - Every token ends with a field splitter or the string end
- This is exactly how newlines behave, although it can be surprising

regex_token_iterator: Field Splitting (2/2)

```
const regex r("^\\s+|\\s*,\\s*|\\s+$");
const string s("  ape,bat, cat ,dog , emu, fox hound  ");
for (sregex_token_iterator i(
    s.begin(), s.end(), r, -1), end; i != end; ++i) {
    cout << i->length() << " (" << *i << ")" << endl;
}
```

- Prints:

```
0 ()
3 (ape)
3 (bat)
3 (cat)
3 (dog)
3 (emu)
9 (fox hound)
```

Questions?

- My E-mail address: stl@microsoft.com
- For more information, see:
  - The current Working Paper:
    http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3376.pdf
  - The C++ Standard Library Extensions: A Tutorial And Reference by Pete Becker