C++11 Regular-Expression Library

By Brian Overland

Jun 25, 2013

This chapter is from the book

C++ for the Impatient

Learn More Buy

Exercises

1. The following string pattern almost works but fails to separate numbers from operators in some cases. (Fortunately, the sample application contains the correct pattern.) Without looking at the application, determine what's wrong with the following regular expression:

```
[0-9]+(.[0-9]*)?
```

2. Add support for a leading minus sign so that "-100", for example, is interpreted as a negative number. Ambiguities arise if the minus sign (-) is simply a unary operator in addition to being a binary operator (which it already is). But if you make the minus sign an optional part of a digit string, the user can add spaces for clarity as needed. Questions: Why does this exercise depend on the **regex** object looking for a digit string first and operators second? What would happen if it looked for operators first?
3. In the sample application, numbers of the form "33" and "33.03" are both accepted; "0.333" and "3." are also. However, strings with no integer portion and no leading zero—such as ".333"—are not accepted. Revise the expression for number_pattern so that it accepts digit strings such as ".333" in addition to the other formats.
4. Extend the floating-point format even further, to accept the following scientific-notation formats:

```
digitsEdigits
digits.digitsEdigits
```

5. Each time an operator is processed, the application assumes there are at least two values on the stack. If there are less than two, the user has entered too many operators. Revise the application to handle this situation by reporting the error and then continuing rather than incorrectly popping the stack (which causes program failure if you let it happen). Hint: Check the size of the stack.
6. Add other operators, such as a binary ^ operator, to perform exponentiation: "2 3 ^" should produce 2 to the 3rd power. Also add the tilde (~) as a unary operator to perform arithmetic negation (multiplying by -1). Finally, add the == and != operators to perform test-for-equality and test-for-inequality. These operators, which are Boolean, should each return either 1 or 0.
7. As a *really* advanced exercise (not for the faint of heart), enable the RPN calculator to read symbols and assign values by using a single equal sign (=) as an assignment operator. Build a symbol table by using the **map** template. Whenever a symbol—that is, a name—appears in any context other than the left side of an assignment, look up its value in the map.

Share This Save To Your Account < Back **Page 10** of 10