C++11 Regular-Expression Library

By Brian Overland
Jun 25, 2013

This chapter is from the book

20.9. Sample App: RPN Calculator

In *C++ Without Fear,* 2nd Edition (Prentice Hall), I presented a Reverse Polish Notation (RPN) calculator as one of the more advanced examples. In this section, I present a superior version of that app.

An RPN calculator lets the user enter arbitrarily long arithmetic expressions in postfix notation. For example, to add 3 and 4, you specify not “3 + 4” but:

```
3 4 +
```

This might at first seem counterintuitive until you realize it’s an elegant notational system that does away with the need for parentheses. For example, the RPN expression:

```
3 4 + 10 1.5 + *
```

is equivalent to the following standard (infix) expression:

```
(3 + 4) * (10 + 1.5)
```

With RPN, an operator always applies to the expressions that precede it. In this case, the asterisk (*) applies to the expressions “3 4 +” and “10 1.5 +”, which produce 7 and 11.5, respectively. Multiplication is finally applied to produce 80.5. The RPN grammar can be summarized as:

```
expression  <=  number
expression  <=  expression expression op
```

Calculations: The stack mechanism, described in Section 16.3, “The stack Template,” is what powers this application. When the program reads a number, it pushes that number onto the stack. When the program reads an operator, it pops the top two values off the stack, performs a calculation, and pushes the result back onto the stack.

Lexical analysis: It’s easy enough to interpret a line of input in which spaces separate operators as well as numbers. The more challenging problem is to recognize operators as both tokens *and* separators so that some of the spaces are optional. For example, it would be desirable to interpret:

```
3 44*5  1.2+/
```

as if it were written as:

```
3 44 * 5 1.2 + /
```

The **strtok** function is inadequate for this task. So is the **regex_token_iterator** function. The solution is to use a **regex_iterator** and search for sequences of characters that constitute either of the following:

A number, consisting of consecutive digits with or without a fractional portion, such as “3”, “44”, or “100.507”

Any of several operators: +, -, *, or /

With this approach, it’s not necessary to have spaces on either side of an operator, although spaces are freely permitted. The following will work just fine:

```
3 4+ 1 2+*
```

The code for the application follows.

```
#include <iostream>
#include <string>
#include <cctype>
#include <regex>
#include <stack>

using std::cout;         // Alternatively, you can use
using std::cin;          //  using namespace std;
using std::endl;
using std::string;
using std::regex;
using std::sregex_iterator;
using std::stack;

void process_token(string s);
stack<double> st;

main() {
     string instr;
     string num_pattern("[0-9]+(\\.[0-9]*)?");
     string op_pattern("[+*/-]");
     regex re(num_pattern + "|" + op_pattern);
     while (true) {
         cout << "Enter expression (or ENTER to exit): ";
         getline(cin, instr);
         if (instr.length() == 0) { break; }
         sregex_iterator it(instr.begin(), instr.end(), re);
         sregex_iterator it_end;
         for (;it != it_end; ++it) {
              process_token(it->str());
         }
         if (!st.empty()) {
              cout << "The value is: " << st.top() << endl;
         }
     };
     return 0;
}

void process_token(string s) {
     // If s contains any char that is NOT an op,
     //  consider it a number by default.
     if (s.find_first_not_of("+*/-") != s.npos) {
          st.push(atof(s.c_str()));
     } else {
          double op2 = st.top(); st.pop();
          double op1 = st.top(); st.pop();
          switch(s[0]) {
          case '+': st.push(op1 + op2); break;
          case '-': st.push(op1 - op2); break;
          case '*': st.push(op1 * op2); break;
          case '/': st.push(op1 / op2); break;
          }
     }
}
```

This program illustrates a couple of important features of regular-expression grammar. First, as mentioned earlier, special characters such as “*” and “+” do not need to be escaped when they occur inside brackets (although brackets themselves would need to be escaped to be treated literally, of course). Also, the minus sign (-) does not need to be escaped, because it does not occur between two other characters.

```
     string op_pattern("[+*/-]");
```

Another interesting feature is that order is potentially significant. The **regex** object in this application searches for a digit string first and *then* for an operator. This order makes Exercise 2 possible.

Share This   Save To Your Account   < Back **Page 9** of 10 Next >