sec06_02/calculator.cpp

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

using namespace std;

/*
   This calculator uses the reverse Polish notation.
*/
int main()
{
   stack<int> results;
   cout << "Enter numbers and operators, separated by spaces, " << endl
      << "P to print, Q to quit." << endl;
   bool done = false;
   while (!done)
   {
      string input;
      cin >> input;

      if (input == "+" || input == "-" || input == "*" || input == "/")
      {
         // If the command is an operator,
         // pop the arguments and push the result

         if (results.size() < 2)
         {
            cout << "Insufficient arguments" << endl;
            return 1;
         }

         // Note that the second argument is on the top of the stack

         int arg2 = results.top();
         results.pop();
         int arg1 = results.top();
         results.pop();

         if (input == "+")
         {
```

```
41             results.push(arg1 + arg2);
42          }
43          else if (input == "-")
44          {
45             results.push(arg1 - arg2);
46          }
47          else if (input == "*")
48          {
49             results.push(arg1 * arg2);
50          }
51          else
52          {
53             results.push(arg1 / arg2);
54          }
55       }
56       else if (input == "Q" || input == "q"
57          || input == "P" || input == "p")
58       {
59          if (results.size() > 0)
60          {
61             cout << results.top() << endl;
62          }
63          if (input == "Q" || input == "q")
64          {
65             done = true;
66          }
67       }
68       else
69       {
70          // Not an operator--push the input value
71
72          results.push(stoi(input));
73       }
74    }
75    return 0;
76 }
```

14.6.3 Evaluating Algebraic Expressions

Using two stacks, you can evaluate expressions in standard algebraic notation.

In the preceding section, you saw how to evaluate expressions in reverse Polish notation, using a single stack. If you haven't found that notation attractive, you will be glad to know that one can evaluate an expression in the standard algebraic notation using two stacks—one for numbers and one for operators.

First, consider a simple example, the expression 3 + 4. We push the numbers on the number stack and the operators on the operator stack. Then we pop both numbers and the operator, combine the numbers with the operator, and push the result.

	Number stack Empty	Operator stack Empty	Unprocessed input 3 + 4	Comments
1	3		+ 4	
2	3	+	4	
3	4 3	+	No more input	Evaluate the top.
4	7			The result is 7.

This operation is fundamental to the algorithm. We call it "evaluating the top".

In algebraic notation, each operator has a *precedence*. The + and - operators have the lowest precedence, * and / have a higher (and equal) precedence.

Consider the expression 3 × 4 + 5. Here are the first processing steps:

	Number stack Empty	Operator stack Empty	Unprocessed input 3 × 4 + 5	Comments
1	3		× 4 + 5	
2	3	×	4 + 5	
3	4 3	×	+ 5	Evaluate × before +.

Because × has a higher precedence than +, we are ready to evaluate the top:

	Number stack	Operator stack		Comments
4	12	+	5	
5	5 12	+	No more input	Evaluate the top.
6	17			That is the result.

With the expression, 3 + 4 × 5, we add × to the operator stack because we must first read the next number; then we can evaluate × and then the +:

	Number stack Empty	Operator stack Empty	Unprocessed input 3 + 4 × 5	Comments
1	3		+ 4 × 5	
2	3	+	4 + 5	

3	4 3	+	× 5	Don't evaluate + yet.
4	4 3	× +	5	

In other words, we keep operators on the stack until they are ready to be evaluated. Here is the remainder of the computation:

	Number stack	Operator stack		Comments
5	5 4 3	× +	No more input	Evaluate the top.
6	20 3	+		Evaluate top again.
7	23			That is the result.

To see how parentheses are handled, consider the expression 3 × (4 + 5). A (is pushed on the operator stack. The + is pushed as well. When we encounter the), we know that we are ready to evaluate the top until the matching (reappears:

	Number stack Empty	Operator stack Empty	Unprocessed input 3 × (4 + 5)	Comments
1	3		× (4 + 5)	
2	3	×	(4 + 5)	
3	3	(×	4 + 5)	Don't evaluate × yet.
4	4 3	(×	+ 5)	
5	4 3	+ (×	5)	
6	5 4 3	+ (×	)	Evaluate the top.
7	9 3	(×	No more input	Pop (.
8	9 3	×		Evaluate top again.
9	27			That is the result.

Here is the algorithm:

```
If you read a number
   Push it on the number stack.
Else if you read a (
   Push it on the operator stack.
Else if you read an operator op
   While precedence(top of operator stack) ≥ precedence(op)
      Evaluate the top.
   Push op on the operator stack.
Else if you read a )
   While the top of the stack is not a (
      Evaluate the top.
   Pop the (.
Else if there is no more input
   While the operator stack is not empty
      Evaluate the top.
```

At the end, the remaining value on the number stack is the value of the expression.

The algorithm makes use of this helper member function that evaluates the topmost operator with the topmost numbers:

```
Evaluate the top:
Pop two numbers off the number stack.
Pop an operator off the operator stack.
Combine the numbers with that operator.
Push the result on the number stack.
```