```
//files and namespace
#include <iostream>
#include <stack>
#include <cmath>
using namespace std;

//prototype
double handleExpression(string exp);
void getExpression();

//Parenthesis handleing
//mainly for uneven parenthesis
stack<char> parenHandleing (stack<char> operators){
  int open = 0;
  int close = 0;
  stack<char> temp;
  //checks number of open and close parenthesis
  while(!operators.empty()){
    char c = operators.top();
    operators.pop();
    if (c == '('){
      open++;
    } else if (c == ')'){
      close++;
    }
    temp.push(c);
  }
  //if there are more close than open
  if (open < close){
    int diff = close - open;
    for (int i = 0; i < diff; i++){
      temp.push('(');
    }
  }
  temp.push('(');
  while(!temp.empty()){
    operators.push(temp.top());
    temp.pop();
  }
  //if there are more open than close
  if (close < open){
    int diff = open - close;
    for (int i = 0; i < diff; i++){
      operators.push(')');
    }
  }
  operators.push(')');
  while(!operators.empty()){
      temp.push(operators.top());
      operators.pop();
    }
  return temp;
}

//do the math
double applyOperator(double a, double b, char op) {
  switch (op) {
    case '+': return a + b;
    case '-': return a - b;
    case '*': return a * b;
    case '/':
      if (b != 0 ){
        return a / b;
      }else {
        cout << "This results in an invalid math operation." << endl << endl;
        getExpression();
      }
    case '%': return fmod(a, b);
      if (b != 0 ){
        return a / b;
      }else {
        cout << "This results in an invalid math operation." << endl << endl;
        getExpression();
      }
    case '^': return pow(a, b);
    default: return 0.0; // Handle unsupported operators
  }
}

//order of operations
//greater the number, the mroe important the operation is
int precedence(char op) {
  if (op == '^') return 3;
  if (op == '*' || op == '/' || op == '%') return 2;
  if (op == '+' || op == '-') return 1;
  return 0;
}
```

```
//evaluates the expression
double evaluateExpression(stack<char> expression){
  stack<double> values;
  stack<char> operators;
  while(!expression.empty()){
    //gets char from expression stack
    char c = expression.top();
    expression.pop();
    //if c is a number
    if (isdigit(c)) {
      int num = c - '0';
      //if the number is multi digit
      while (!expression.empty() && isdigit(expression.top())){
        num = (10*(num)) + (expression.top()-'0');
        expression.pop();
      }
      values.push(num);
    // if c is (
    } else if (c == '(') {
      operators.push(c);
      //if there is a negative number
      if (!expression.empty() && expression.top() == '-') {
        expression.pop();
        if (isdigit(expression.top())) {
          c = expression.top();
          expression.pop();
          int num = -(c - '0');
          while (!expression.empty() && isdigit(expression.top())){
            num = (10*(num)) + (expression.top()-'0');
            expression.pop();
          }
          values.push(num);
        } else if (!expression.empty() && expression.top() == '(' && values.empty()) {
          values.push(0);
          operators.push('-');
        } else {
          cout << "There is something wrong with this expression. Please try again." << endl
<<endl;
          getExpression();
        }
      }
    // if c is )
    } else if (c == ')') {
      //if the parenthesis is not empty
      while (!operators.empty() && operators.top() != '(') {
        double b = values.top(); values.pop();
        double a = values.top(); values.pop();
        char op = operators.top(); operators.pop();
        values.push(applyOperator(a, b, op));
      }
      operators.pop();
    //if c is an operator
    } else if (c == '+' || c == '-' || c == '*' || c == '/' || c == '%' || c == '^') {
      // if there is another operator already on the stack
      // and the precedence of the new operator is less than or equal to the precedence of the
operator on the stack
      if (values.empty() && !expression.empty() && (expression.top() == '(' ||
isdigit(expression.top()))){
        values.push(0);
      } else if (!values.empty()){
        // if the stack is empty or the precedence of the new operator is greater than the
precedence of the operator on the stack
        while (!operators.empty() && precedence(operators.top()) >= precedence(c)) {
          double b = values.top(); values.pop();
          double a = values.top(); values.pop();
          char op = operators.top(); operators.pop();
          values.push(applyOperator(a, b, op));
        }
      } else {
        cout << "There is something wrong with this expression. Please try again." << endl <<endl;
        getExpression();
      }
      operators.push(c);
    }
  }
  // if there are still operators on the stack
  while (!operators.empty()) {
    double b = values.top(); values.pop();
    double a = values.top(); values.pop();
    char op = operators.top(); operators.pop();
    values.push(applyOperator(a, b, op));
  }
  return values.top();
}

//gets expression from user or 'done' if the user is done with the calculator
//checks if input is empty, done, or something to pass to handleExpression()
void getExpression(){
  while (true){
    string expression;
    cout << "Enter an arithmetic expression: ";
    getline(cin, expression);

    if (!expression.empty()){
      if (expression == "done"){
        cout << "Thank you for using the calculator!" << endl;
        exit(0);
      } else{
        double result = handleExpression(expression);
        cout << "Result: " << result << endl;
      }
    } else{
      cout << "Please enter an expression or 'done' if you are done." << endl;
    }
    cout << endl;
  }
}

//once an expression has been given, handleExpression checks for valid syntax
//and puts it in a stack
double handleExpression(string exp) {
  stack<char> expression;
  for (char c : exp) {
    if (isdigit(c) || c == '+' || c == '-' || c == '*' || c == '/' || c == '%' || c == '^' || c == '(' || c == ')' || c == '
'){
      if (c != ' '){
        expression.push(c);
      }
    } else{
      cout << "Invalid expression. Only numeric expressions are allowed." << endl <<endl;
      getExpression();
    }
  }
  expression = parenHandleing(expression);
  double result = evaluateExpression(expression);
  return result;
}

//main function
//welcomes user to calculator and calls getExpression()
int main() {
  cout << "Welcome to the calculator!" << endl << "If you wish to use negative numbers, please
put all negative numbers in their own parentheses. Ex. (-34)" << endl << "To end the calculator
enter 'done' in all lower case." << endl <<endl;
  getExpression();
  return 0;
}
```