```
#include <iostream>
#include <cstring>
#include <cmath>

using std::cout;
using std::cin;
using std::endl;
using std::cerr;

// Function to eliminate spaces from a string
void eliminateSpaces(char* expr);

// Function to evaluate mathematical expression
double evaluateExpression(char* expr);

// Function to get the next term  starting from the position index in the expression
double getTerm(char* expr, int& index);

// Function to get the value of a single number at index position in the expression
double getNumber(char* expr, int& index);

// Function to extract sub expression within parentheses
char* extractSubExpression(char* expr, int& index);


int main() {

    // Maximum expression length including terminating \0 character
    const int MAX(80);

    // String containing expression to be evaluated
    char inputExp[MAX];

    // Start the program by greeting the User
    cout << "Welcome to the calculator of 1901.5" << endl
         << "Enter an expression to evaluate, or an empty line to exit." << endl;


    // Take input in an infinite loop. loop will be exited based on the user input
    for(;;)
    {
        // add a new line and show input marker
        cout << endl;
        cout << ">> ";

        // input expression to be evaluated
        cin.getline(inputExp, sizeof(inputExp));

        // Eliminate spaces from the input string
        eliminateSpaces(inputExp);

        // if input string in an emply line, exit the calculator
        if(!inputExp[0]) return 0;

        // Try to evaluate the expression and output the result
        try
        {
            double result(0);
            result = evaluateExpression(inputExp);

            // show output marker and display the result
            cout << ">> = " << result << endl;
        }
        // In case of any errors, print the error message
        catch (const char* errMsg)
        {
            cerr << ">> Error: " << errMsg << endl;
        }

    }

    // Return to OS
    return 0;
```

```
}

// Function to eliminate spaces from a string
void eliminateSpaces(char* expr)
{
    // Position in the actual expression
    int posActual(0);

    // Position in the modified expression
    int posModified(0);

    // scan the expression until \0 is found
    while(expr[posActual] != '\0')
    {
        // if the character at position posActual is not a space then move this character to position
        // posModified and increment both posActual and posModified
        if (expr[posActual] != ' ')
        {
            expr[posModified] = expr[posActual];
            posActual++;
            posModified++;
        }
        // Else if a space is found, increment only the posActual
        else
            posActual++;

    }

    // Append \0 to end of modified expression
    expr[posModified] = '\0';

}

// Function to evaluate mathematical expression
double evaluateExpression(char* expr)
{
    // Evaluated value of the expression
    double exprValue(0);

    // index to keep track of the current position in the expression being evaluated
    int index(0);

    // Get the first term in the expression
    exprValue = getTerm(expr, index);

    // scan through the remaining expression and add, subtract as required
    // using an indefinite loop, which will exit when the \0 character is found
    // or an error occurs
    for(;;)
    {
        // Choose the required process based on the current position in the expression
        switch(expr[index])
        {
            // if the expression has reached to terminating character, return the value of expression
            case '\0':
                return exprValue;
                break;

            // if the current character is +, increase the value of index variable and add the next term
            // to the current value of expression
            case '+':
                exprValue += getTerm(expr, ++index);
                break;

            // if the current character is -, increase the value of index variable and subtract the next↙
     term
            // from the current value of expression
            case '-':
                exprValue -= getTerm(expr, ++index);
                break;

            // if none of the above cases matches the current character, expression cannot be evaluated
```

```
            // throw an error message
            default:
                char errMsg[38] = "Expression evaluation error. Found: ";
                // append the un-understood character to the error message
                strncat_s(errMsg, expr + index ,1);
                // throw the error message
                throw errMsg;
                break;

        }

    }

}

// Function to get the next term  starting from the position index in the expression
// index is passed as reference so that while evaluating the term, index value
// from function evaluateExpression can be moved onward
double getTerm(char* expr, int& index)
{
    // Evaluated value of the term
    double termValue(0);

    // set the term value equal to first number in the term
    termValue = getNumber(expr, index);

    // scan through the remaining expression and multiply, divide as required
    // using an indefinite loop, which will exit when the next operator is not
    // either of the following: *(product), /(division)

    for(;;)
    {
        // Choose the required process based on the character at current position in the expression
        switch(expr[index])
        {
            // if the current character is *, increase the value of index variable and multiply the next↙
     number
            // with the current value of term
            case '*':
                termValue *= getNumber(expr, ++index);
                break;

            // if the current character is /, increase the value of index variable and divide
            // the current value of term with the next number
            case '/':
                termValue /= getNumber(expr, ++index);
                break;

            // if neither of above operators are found, consider the term ended and return the value of ↙
    the term
            default:
                return termValue;
                break;

        }
    }
}


// Function to get the value of a single number at index position in the expression
double getNumber(char* expr, int& index)
{
    // Evaluated value of the number
    double numberValue(0);

    // If character at current index is '(' then extract the sub-expression between parentheses and
    // evaluate the sub-expression first and consider the evaluated value as the value of number
    // check if there is a power specified to this number i.e. Go directly to power checking lines of
    // code
    if(expr[index] == '(')
    {
        char* subExpr = nullptr;
        subExpr = extractSubExpression(expr, ++index);
```

```
        numberValue = evaluateExpression(subExpr);
        // skip the steps to form number from the digits and go directly to steps to raise the number to
        // the specified power
        goto CheckForPower;
    }

    // if the flow reaches this point, there is no parenthesis encountered
    // There must be at least one digit to form a number. If not so, throw an error message
    if(!(isdigit(expr[index])))
    {
        char errMsg[31] = "Invalid character in number: ";
        // append the problematic character to error message
        strncat_s(errMsg,expr+index, 1);
        throw errMsg;
    }

    // Loop to accumulate the digits leading the decimal point
    // As long as a digit is found
    // multiply the current value of number by 10 and
    // add (the ascii value of current digit - the ascii value of digit zero) to the value of number
    while (isdigit(expr[index]))
        numberValue = 10*numberValue + (expr[index++] - '0');

    // now check if the next digit is a decimal point
    // if so, loop for the digits following the decimal point and add their value to number
    if ('.' == expr[index])
    {
        // Move the index to the next character
        index++;

        // loop for digits following the decimal point
        // Place value factor
        double factor(1.0);

        // Loop as long as we have any digits
        // Modify the place value factor by dividing by 10
        // add the (value of current digit * place value factor) to the current value of number
        while(isdigit(expr[index]))
        {
            factor/=10.0;
            numberValue += (expr[index++] - '0')*factor;
        }

        // when no more digits are found, number value is obtained
        // carry on checking for power character
    }

    CheckForPower:
    // Check if the next character is exponent '^'
    // if so, raise the numberValue to power specified
    if ('^' == expr[index])
    {
        // obtain the value of number representing the power to be raised
        double power = getNumber(expr, ++index);

        // raise the number value to the required power
        numberValue = pow(numberValue,power);
    }

    // Finally return the value of the number
    return numberValue;

}

// Function to extract sub expression within parentheses
char* extractSubExpression(char* expr, int& index)
{
    // Pointer to sub expression to return
    char* psubExpr(nullptr);

    // Count of Left parentheses encountered
    int countLeft(0);
```

```
    // Saved Starting Value of index
    int startIndex(index);

    // Loop until the expression ends or a corresponding right parenthesis is found
    do
    {
        // only if a left or right parenthesis is found, do the following actions
        // else continue looping the characters
        switch (expr[index])
        {
            // if another left parenthesis is found, increase the countLeft
            case '(':
                countLeft++;
                break;
            // if right parenthesis is found
            case ')':
                // check if no additional left parenthesis have been found in the sub expression...
                // or when a corresponding right parenthesis have been found for all additional...
                // left ones, i.e. countLeft is zero
                // then increase the index
                // allocate memory for sub expression having length (index - startIndex)...
                // index was raised in previous line to accomodate \0 in place of ')'
                // check if memory allocation failed, throw an exception
                // copy the subexpression in the memory pointed by pSubExpr
                // return subexpression pointer
                // --- Else ---
                // reduce the count of left parenthesis to be matched
                if (countLeft == 0)
                {
                    // raise index
                    ++index;

                    // allocate memory
                    psubExpr = new char(index - startIndex);

                    // check if allocation successful ?
                    if(!psubExpr)
                        throw "Memory allocation failed.";

                    // copy sub expression to pSubExpr
                    // copy from (expr starting from startindex) to psubExpr
                    // number of bytes to copy = (index - startIndex -1)
                    // size of psubExpr in which the chars are to be copied = (index - startIndex)
                    strncpy_s(psubExpr, (index-startIndex), (expr+startIndex), (index-startIndex-1));

                    // return sub expression
                    return psubExpr;
                }
                else
                    // reduce count of Left Parenthesis
                    --countLeft;

                // end the case of right parenthesis
                break;

        } // end switch

    } while(expr[index++] != '\0');

    // if the flow reaches this point, means that \0 character encountered before all left parenthesis..↙
    .
    // found a matching right one...
    // there must be some error, so throw error message
    throw "Not all opening parentheses found a matching closing parenthesis.";

}
```