CS205 Project 1: A Simple Calculator

Yang Yanzhuo
12212726

INTRODUCTION

This project aims to develop a simple calculator that computes the sum, difference, product, and quotient of two numbers when the input is correctly formatted. If the input is invalid, the calculator will output error messages to guide the user to provide proper inputs.

This calculator is also equipped with these advanced features:

1. **Run in Single mode**, which means the program will **automatically end** after program **only one** group of data.
2. **Run in Interactive mode**, which means the user can calculate **multiple pair of numbers** and will **not end until inputting** `quit` in the command line.
3. **Multiplication of Two Large Integers**: It can handle the multiplication of large integers that exceed the typical value range for standard data types.
4. **Scientific Notation**: The calculator accepts inputs in scientific notation with output in scientific notation.

CLAIM

1. When you want to input from the command line arguments, please use `'*'` instead of `*`.
2. Please add the minus sign directly before the negative number without any braces.
3. When you do the big integer multiplication in interactive mode, pay attention the the space between operands and operator is required.
    - For instance, `123 * 321` is accepted, while `123*321` not.
4. The grammar and sentence structure of both the report and the code have been optimized and polished by using ChatGPT 4.0 (OpenAI, 2024-03-10).
5. Development Environment:
    - `Ubuntu 20.04.2 LTS x86_64` with gcc version `gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0`
    - C standard: `C11`

IMPLEMENTATION

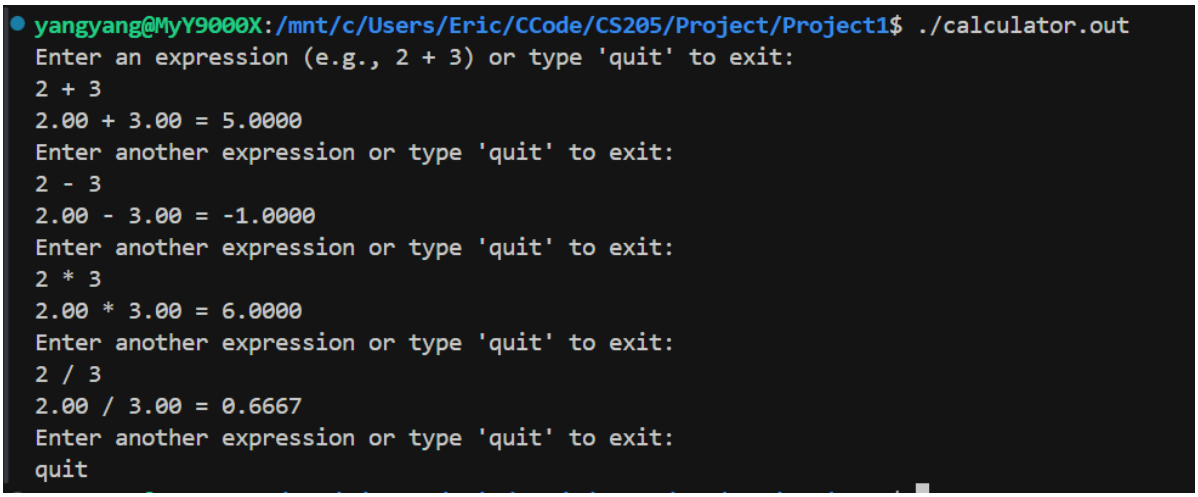
1. Check the number of arguments to select SINGLE or INTERACTIVE mode

Read `argc`, which stores the number of arguments from the command line.

If the `argc` is **exactly one** which means the user inputs only one arguments which is the name and the path of the program or executable file, then **enter the interactive mode**.

Type `quit` to terminate the program.

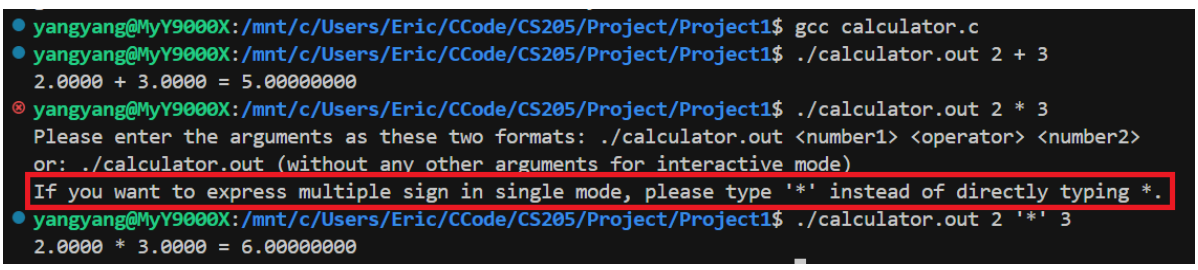
Test Cases:



If the `argc` is **exactly four**, which means the user input one operator between two operands after the name of the executable file, then **enter the single mode**.

Test Cases:

(Here we use `'*'` instead of `*` as claimed)



Otherwise, the program will output error message to guide the user to provide proper inputs, just like the third command in the picture above shows,

---

Implementation: (Only key part)

```
double calculate(double operand1, char operators, double operand2)
{   // Get the answer by directly using the operator.
    switch (operators) {
        case '+':
            return operand1 + operand2;
        case '-':
            return operand1 - operand2;
        case '*':
            return operand1 * operand2;
        case '/':
            return operand1 / operand2;
        default:
            printf("Error: Invalid operator '%c'.\n", operators);
            return -1;
    }
}
```

```
if (argc == 4) {
    /*Single mode*/
    //
    if (checkDividedZero(operators, operand2))
        printf("Error: Division by zero\n");
    else {
        // Get the answer by directly using the operator.
        result = calculate(operand1, operators, operand2);
        printf("%.2lf %c %.2lf = %.4lf\n", operand1, operators,
operand2, result);
    }

} else if (argc == 1) {
    char input[100];
    /*Interactive mode loop*/
    while (true) {
        fgets(input, 100, stdin);
        if (checkQuit(input))
            break;
        //
        printf("Enter another expression or type 'quit' to
exit:\n");
```

```
    }

} else {
    /*Error message to guide users*/
    //
}
```

2. Check if the input is valid & Check the termination condition

2.1 Check the operator sign

```
bool checkValidOperator(char operators) {
    if ( (operators == '+') || (operators == '-') || (operators == '*') || (operators == '/') )
        return true;
    else
        return false;
}
```

2.2 Check if divided by zero

```
bool checkDividedZero(char operators, double operand2) {
    return (operand2 == 0) && (operators == '/');
}
```

2.3 Check if the user wants to terminate the program in interactive mode

```
bool checkQuit(const char input[]) {  //ASCII value of `quit`.
    return (input[0] == 113) && (input[1] == 117) && (input[2] == 105) && (input[3] == 116);
}
```

2.4 Check if the input number is enough big to change the algorithm

```
bool checkBig(double operand1, double operand2) {  //Double type gradully loss of precision when gets bigger.
    return operand1 * operand2 >= 1e7;
}
```

---

3. Big Integer Multiplication

When it comes to big number multiplication, we only consider the **integer condition** for these reasons:

- The fractional portion has **little effect on accuracy**.
- If precise results is needed, we can get the answer by **manually moving the decimal point**. For instance, we want to know the precise result of $6.6 * 8.8$:
  - $6.6 * 8.8 = ?$
  - Consider $66 * 88$, and get 5808
  - Then **manually move** (by user himself) the decimal point to the correct position: 58.08

Here is how we deal with the big integer multiplication:

We still use the example that $66 * 88$:

```
      6 6
*     8 8
---------
    5 2 8
+ 5 2 8
---------
= 5 8 0 8
```

This process actually uses the principle of **Column Multiplication**.

Here provides its principle, which involves multiplying the **digits** of two numbers and adding them according to their **place value** to get the final result.

Suppose we want to get $a * b$, and expand `a` and `b`

$$a = (a_n a_{n-1} \dots a_1 a_0)_{10} = a_n 10^n + a_{n-1} 10^{n-1} + \dots + a_1 10 + a_0$$

$$b = (b_m b_{m-1} \dots b_1 b_0)_{10} = b_m 10^m + b_{m-1} 10^{m-1} + \dots + b_1 10 + b_0$$

Then,

$$a \times b = \sum_{j=0}^{m} \left(b_j 10^j \times \sum_{i=0}^{n} a_i 10^i \right)$$

During the process, we can easily find that it is troublesome to deal with both multiplication and addition. Thus, we optimize the process of addition, **forget about the carry** at the beginning:

```
        6     6
*       8     8
-----------------
       (48) (48)
+ (48) (48)
-----------------
= (48) (96) (48)
```

Then, add the numbers from right to left: if the number at a given position is greater than 10,

- Use **the modulo ten operation** and keep **the reminder result** on that position;
- Carry over **the quotation resulter** (obtained through the modulo operation) to the next higher position;
- **Repeating** this process until the addition is complete.

Finish the multiplication according to the optimized algorithm:

```
  (48) (96) (48)  ---------------- 48  / 10 = 4  ... 8
  (48) (100) (8)  ---------------- 100 / 10 = 10 ... 0
  (58) (0) (8)    ---------------- 58  / 10 = 5  ... 8
  (5) (8) (0) (8) ================ DONE
```

Implementation

STEP 0: Check if the number is enough big.

```
bool checkBig(double operand1, double operand2) {
    return operand1 * operand2 >= 1e7;
}
```

STEP 1: Transfer the double data type into array type. Every digit takes place one element in array in order.

The higher in array, the lower in the original integer.

> If the operand = 456,
> then we will get:
> a[0] = 4;
> a[1] = 5;
> a[2] = 6;

```
int getLength(double operand) {   // Get the length of the array.
    int count = 1;

    if (operand == 0)
        return 1;

    while (operand >= 10) {
        operand /= 10;
        count++;
    }
    return count;
}
```

In Interactive mode:

```
int sizeA = getLength(operand1);
int sizeB = getLength(operand2);
int a[sizeA], b[sizeB];
for (int i = 0; i < sizeA; i++)   // Load array a and b from the original input.
    a[i] = input[i] - 48;         // The difference between ASCII value and digit is 48.
for (int i = sizeA + 3; i < sizeB + sizeA + 3; i++) // We have two spaces in the input.
    b[i - sizeA - 3] = input[i] - 48;
int mulRe[sizeA + sizeB];
```

In single mode:

```
int sizeA = getLength(operand1);
int sizeB = getLength(operand2);
int a[sizeA], b[sizeB];
for (int i = 0; i < sizeA; i++)   // Load array a and b from the command line.
    a[i] = argv[1][i] - 48;       // The difference between ASCII value and digit is 48.
for (int i = 0; i < sizeB; i++)
    b[i] = argv[3][i] - 48;
int mulRe[sizeA + sizeB];
```

STEP 2: Do the optimized multiplication.

```
void bigMul(const int a[], const int b[], int mulResult[], const int sizeA, const int sizeB) {

    for (int i = 0; i < sizeA + sizeB; i++)
        mulResult[i] = 0;

    for (int i = 0; i < sizeA; i++) {
        for (int j = 0; j < sizeB; j++)
            mulResult[i + j + 1] += a[i] * b[j];
    }

    for (int k = sizeA + sizeB - 1; k > 0; k--) {
        if (mulResult[k] >= 10) {
            mulResult[k - 1] += mulResult[k] / 10;   // Add with the quotation.
            mulResult[k] %= 10;    // Store with the reminder
        }
    }

}
```

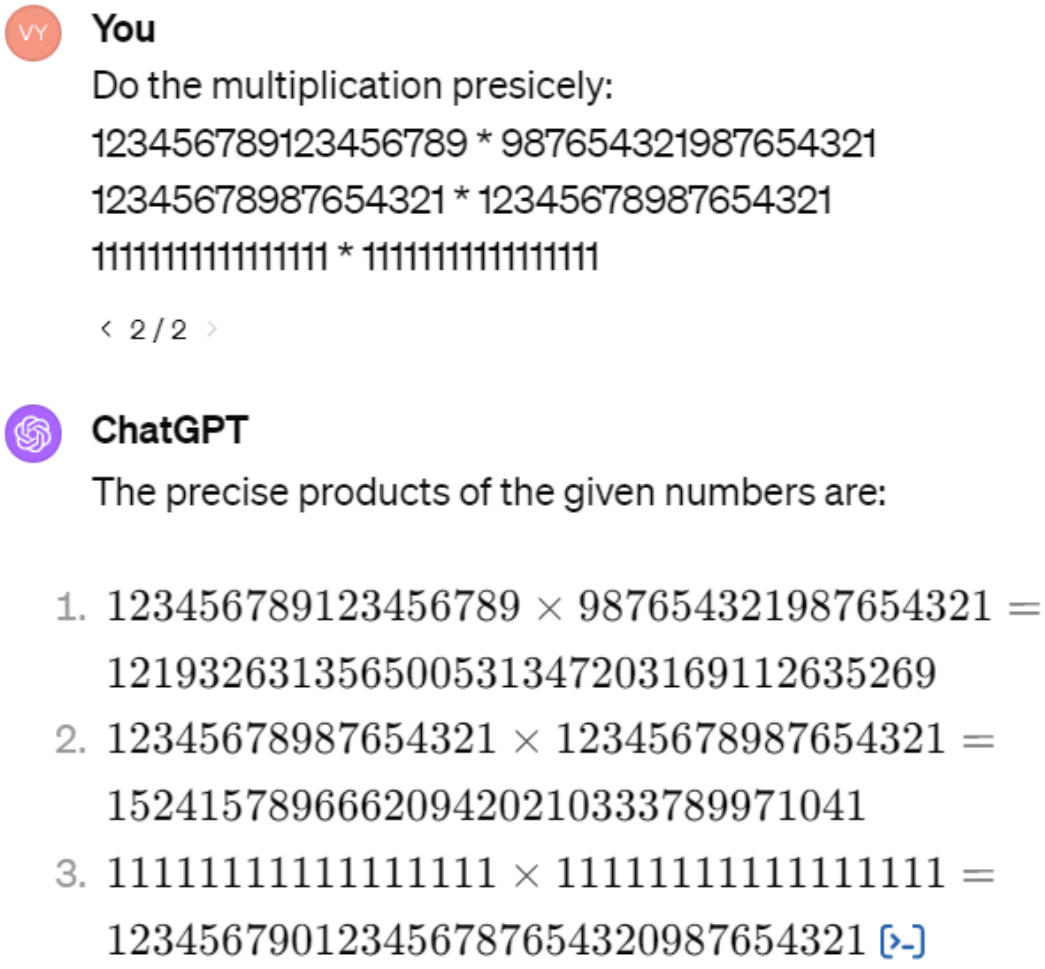
STEP 3: Implement it in the `main` function

```
bigMul(a, b, mulRe, sizeA, sizeB);
for (int i = 0; i < sizeA; i++)
    printf("%d",a[i]);
printf(" * ");
for (int i = 0; i < sizeB; i++)
    printf("%d",b[i]);
printf(" = ");
    for (int i = 0; i < sizeA + sizeB; i++) {
        if (i == 0 && mulRe[i] == 0)
            continue;
        printf("%d", mulRe[i]);
    }
printf("\n");
```

Test Cases I:

```
yangyang@MyY9000X:/mnt/c/Users/Eric/CCode/CS205/Project/Project1$ ./calculator.out 123456789123456789 '*' 987654321987654321
123456789123456789 * 987654321987654321 = 121932631356500531347203169112635269
yangyang@MyY9000X:/mnt/c/Users/Eric/CCode/CS205/Project/Project1$ ./calculator.out
Enter an expression (e.g., 2 + 3) or type 'quit' to exit:
123456789123456789 * 987654321987654321
123456789123456789 * 987654321987654321 = 121932631356500531347203169112635269
Enter another expression or type 'quit' to exit:
12345678987654321 * 12345678987654321
12345678987654321 * 12345678987654321 = 152415789666209420210333789971041
Enter another expression or type 'quit' to exit:
1111111111111111 * 1111111111111111
1111111111111111 * 1111111111111111 = 1234567901234567876543209876543 21
```

Correctness Check: By ChatGPT 4.0 *(OpenAI, 2024-03-10)* and a simple Python program



-
```python
a = 123456789123456789;
b = 987654321987654321;
print(a * b);

c = 12345678987654321;
d = 12345678987654321;
print(c * d);

e = 1111111111111111;
f = 1111111111111111;
print(e * f);
```

```
C:\Users\Eric\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\Eric\PycharmProjects\pythonProject\.venv\Scripts\test.py
121932631356500531347203169112635269
152415789666209420210333789971041
123456790123456787654320987654321
```

Test Cases II:

```
yangyang@MyY9000X:/mnt/c/Users/Eric/CCode/CS205/Project/Project1$ ./calculator.out
Enter an expression (e.g., 2 + 3) or type 'quit' to exit:
987654321 * 987654321
987654321 * 987654321 = 975461057789971041
```

Correctness Check II: By Shiqi Yu's CS205 project assignment document, released on SUSTech Blackboard *(2024 Spring)*

```
./calculator 987654321 * 987654321 # The result should be
975461057789971041
```

CONCLUSION

---

1. LIMITATION

- Unable to solve with the fractional part **automatically**. But due to our analysis, it is meaningless and lack of necessity.
- Unable to solve the **continuous** operation, such as `1231321 * 4165465\n + 4646`.
- The optimization of big integer multiplication doesn't reach the peak. In other words, there are a lot of parts which could be optimized to make the program faster.

---

2. FEELINGS of CODING

- `C language` works obviously faster than `Java` when I tested the correctness of the program with really large inputs, while the grammar of `Java` is simpler and more readerable.
- I become more familiar with the `C Language` grammar.
- There's still a long way to go for my coding career.