# CS219 Project 1 : A Simple Calculator

## Part I : Basic Informations

### Introduction

This project is mean to develop a simple calculator based on command line. It supports some basic calculations between two numbers, which includes add, subtract, multiply, and divide operations. What's more, the calculator also supports some high level most used functions such as square root, sin, cos functions with one parameter.

To ensure that the calculator can deal with large numbers and high precision calculations, the implemented functions are based on char array parsing and bitwise add/subtractions. In this way, the calculator can deal with any length of numbers as long as it doesn't exceeds the limit number in the setting, which can be modified too.

### User Handbook

The calculator can be used only through command line and has no provided GUI. To open the calculator, first you need to change directory to the file where the `calculator.c` is at, then, do the following to enter the calculation mode:

1. Compile the `calculator.c` into an excutable file:

   ```
   gcc calculator.c -o calculator
   ```

2. In modes privided with arguments, one calculation can be done, and the program will quit immediately. Please be noted that when you are using this mode, symbols like '*' , '(' and ')' needs to add an escape character \ before head, or the computer will treat them as special commands.

   ```
   user % ./caculator 2 + 3
   5
   user % ./calculator sqrt\(9\)
   3
   ```

3. If you want to do more calculations, do not provide the program with arguments

   ```
   user % ./calculator
   >>> 1 + 2
   3
   >>> sqrt(256)
   16
   >>> quit
   user %
   ```

In the second mode, enter `quit` or `exit` to exit the program.

After entering the calculation mode, you can do add, subtraction, multiplication, division and other commonly used functions, but remember that this calculator only supports basic calculations between two numbers, and no parentheses in the expression. Here are the form of expression inputs:

```
add         <symbol>[number1] + <symbol>[number2]
subtract    <symbol>[number1] - <symbol>[number2]
multiply    <symbol>[number1] *(or x) <symbol>[number2]
divide      <symbol>[number1] / <symbol>[number2]
squre root  sqrt([number1])
sin         sin(<symbol>[number1])
```

If you forget it, just enter `help` in the calculation mode and the program will show you the form again. Please note that the symbols are optional, and you can freely decide whether to have spaces between operands and operators (adding space is recommended to generate a less ambiguous expression).

Besides, there are also some macros that are used during the process, they control the max size or precision of some calculations, you are free to modify them to try to get different results.

```
#define OPERAND_MAX_SIZE 200            // max supported size for single operand
#define INUPUT_MAX_SIZE 512             // max supported size for input
#define MAX_E 30                        // the scale threshold for normal print and
scientific notation
#define ANS_PRECISION 5                 // the precision of answer in scientific notation
#define BASE_PRECISION 40               // the basic length of answer in divide operation
#define PI "3.141592653589793238462643" // the value of PI
#define TAYLOR_EXPANSION_MAX_ORDER 10   // the max order or taylor expansion in sin and
cos funtions
#define SIN_PRECISON 8                  // the precision of sin produced answer
```

# Part II : Function Implementation

## Input parsing

Due to the reason that we need to deal with large numbers and high precision numbers that cannot be stored by conventional data structures, we need to parse the input numbers in char array form, so without the basic checkings from the  library functions, it is highly important to ensure our input is valid with multiple layers of checking.

```
int parse_number(const char * input, int i, char * para);
int parse_func(const char * input, int *func_code, char * para);
int parse_expression(const char* input,char* num1,char* op,char* num2,int*
num1_symbol,int* num2_symbol);
int parse_and_calculate(const char* input,char* num1,char* op,char* num2,int*
num1_symbol,int* num2_symbol);
```

Here are the functions that I used for parse the inputs. First, the input enters `parse_and_calculate()`, and we do a basic operation to judge whether the user enters an expression or a function. If it is an expression, goto `parse_expression()` function, if it is a function, goto `parse_func()` function. In both two functions, the operands and parameters need to be parsed into valid char array numbers, so the similar codes are

encapsulated into the `parse_number` function. Here are part of the code:

```
int
parse_number(const char * input, int i, char * para)
{
    /* parameter index */
    int j = 0;
    /* only one dot and e symbol for a number */
    int num_dot_flag = 0;
    int num_e_flag = 0;

    /* parse the parameter */
    while (input[i] != ')' && (isdigit(input[i]) || input[i] == '.' || input[i] == 'e' ||
input[i] == '-' || input[i] == '+'))
    {
        ......
    }
    if(para[j-1]=='e' || para[j-1]=='.')
    {
        printf("number not specified, ");
        return 0;
    }
    para[j] = '\0';
    j = 0;

    /* translate numbers with e symbol into normal representation */
    if(num_e_flag)
    {
        ......
    }

    return i;
}
```

The `parse_number()` function allows numbers both in normal forms and in scientific notations. Symbols in the front of number or just after 'e' notaion is allowed; Decimal point after e noation is not allowed; charactors that is not a number is not allowed, and the number cannot be ended with symbols from{ '.' , 'e' , '-' , '+' }.

After parsing finished, the program will translate the numbers in scientific notations into normal forms to standardize the calculation parameters.

To look into detail, please refer to `calculator.c`.

## Calculation functions

All the calculations are based or indirectly based on char array parsing and bitwise add or subtract operations, so our input parameters are all in char pointer form:

```
char * add(const char * a,const char * b);
char * subtract(const char * a,const char * b, int type);
char * multiply(const char * a,const char * b);
char * divide(const char * a,const char * b);
char * square_root(const char * a);
char * sine(const char * a, int * minus_flag);
```

### Preprocessing

Before every basic calculations(+-*/), we must parse the input numbers and find out their characteristics : Find the integer length and fraction length of each number. Then, to convenient the calculation, we need to store the numbers in another char array without its decimal point. Here are the variables we need to use in the actual calculation, the detailed procedure to get their value is omitted here.

```
/* Record the length of interger parts and farction parts of both number */
int num1_IntegerDigits = 0;
int num1_FractionDigits = 0;
int num2_IntegerDigits = 0;
int num2_FractionDigits = 0;

/* Find the position of '.' */
char * num1_dot = strchr(num1_str, '.');
char * num2_dot = strchr(num2_str, '.');

 /* store the number without its dot */
int num_size = strlen(num1_str) + strlen(num2_str);
char * num1_noDot = malloc(num_size * sizeof(char));
char * num2_noDot = malloc(num_size * sizeof(char));
memset(num1_noDot, 0, num_size * sizeof(char));
memset(num2_noDot, 0, num_size * sizeof(char));
num1_noDot[num_size-1] = '\0';
num2_noDot[num_size-1] = '\0';
```

### add

For add operation, I first found out the offset of the decimal point position between the `num1_noDot` and `num2_noDot`, then, according to the offset, just align the bits and add them from low bits to high bits and store them in an int array `ans_Digits`. At last, traverse the int array to calculate the answer and the carry for each bit.

```
int num1_offset = (int)(max(num1_IntegerDigits,num2_IntegerDigits)) - num1_IntegerDigits;
int num2_offset = (int)(max(num1_IntegerDigits,num2_IntegerDigits)) - num2_IntegerDigits;
j = 0;
for(int i = num1_offset ; i < num1_offset+num1_IntegerDigits+num1_FractionDigits ; i++)
    ans_Digits[i+1] += num1_noDot[j++] - '0';

j = 0;
for(int i = num2_offset ; i < num2_offset+num2_IntegerDigits+num2_FractionDigits ; i++)
    ans_Digits[i+1] += num2_noDot[j++] - '0';
```

```
/* from the lowest bit, add bitwise and calculate the carry */
for(int k = dig_len-1 ; k > 0 ; k--)
{
    int num = ans_Digits[k];
    int carry = num / 10;
    int res = num - carry*10;

    ans_Digits[k] = res;
    ans_Digits[k-1] += carry;
}
```

Because we only traverse every bits of both number for once, so assume two numbers has length of m and n, then the time complexity is O(n + m), which is linear and optimal.

**subtract**

For subtract operation, the theory is similar with the add operation: the key is to align the bits and subtract them from lower bits to higher bits. What's different is that, the bitwise subtraction will fail if the minuend is smaller than the subtrahend. So before we do calculation, we need to compare which number is bigger first, and set it as the minuend. At last, add a minus symbol if we really swap the original order.

```
/* do the subtraction */
for(int k = dig_len-1 ; k >= 0 ; k--)
{
    int num = ans_Digits[k];
    int subtractor = 0;

    if(num2_noDot[k] == 0)
        subtractor = 0;
    else
        subtractor = num2_noDot[k] - '0';

    if(num < subtractor)
    {
        num+=10;
        ans_Digits[k-1]--;
    }
    ans_Digits[k] = num - subtractor;
}
```

What's more about subtraction, because the `divide` function also used the add and subtract functions here, so `subtract` provides two types of forms of answer according to the `type` parameter, if `type==0`, then ouput the normal and standard answer, if `type==1`, then output the answer without changing its length or eliminate its prefix and post fix char zeros.

```
/* produce the answer according to type */
int decimal_pos = (int)max(num1_FractionDigits,num2_FractionDigits);
char *answer;
if(type == 0)
```

```
    answer = trans_to_ans(ans_Digits,dig_len,decimal_pos);
else
{
    answer = malloc((dig_len + 2) * sizeof(char));
    int idx = 0;
    for (int i = 0; i < dig_len; i++)
    {
        if (i == dig_len - decimal_pos)
            answer[idx++] = '.';

        answer[idx++] = ans_Digits[i] + '0';
    }
    if(!idx)
        answer[idx++] = '0';
    answer[idx] = '\0';
}
```

Same as add operation, because we only traverse every bits of both number for once, so assume two numbers has length of m and n, then the time complexity is O(n + m), which is linear and optimal.

**multiply**

multiplication is also done based on bitwise operation, the difference between it and add/subtract is that, we need to calculate all combinations of bits for both numbers, that is to say, a double for loop is needed to calculate every two bits' multiplication, and store them at the right position in the answer integer arrays.

```
/* use double for loop to do the calculation */
for(int i = 0 ; i < num1_IntegerDigits+num1_FractionDigits ; i++)
{
    int num1_cur_digit = num1_noDot[num1_IntegerDigits+num1_FractionDigits - i - 1] - '0';
    for(int j = 0 ; j < num2_IntegerDigits+num2_FractionDigits ; j++)
    {
        int num2_cur_digit = num2_noDot[num2_IntegerDigits+num2_FractionDigits - j - 1] -
'0';
        int ind = num1_IntegerDigits + num2_IntegerDigits + num1_FractionDigits +
num2_FractionDigits - (i+j+1);
        ans_Digits[ind] += num1_cur_digit * num2_cur_digit;
    }
}

/* calculate the number on each bit(might be bigger than 10)
 * and add the carry to next the next bit
 */
for(int k = num1_IntegerDigits+num2_IntegerDigits+num1_FractionDigits+num2_FractionDigits
- 1 ; k > 0 ; k--)
{
    int num = ans_Digits[k];
    int carry = num / 10;
    int res = num - carry*10;

    ans_Digits[k] = res;
    ans_Digits[k-1] += carry;
```

```
}
```

Becaused we used double for loops, so assume two numbers has length of m and n, then the time complexity is O(nm), which is also the the case when we do multiplication by hand.

**divide**

Division is done differently because not like add, subtraction, and multiplication, the bitwise division can't lead to the division of the whole number. So, I roughly think of a basic algorithm that can do it:

```
set quotient as 0
while(dividend is bigger than divisor)
{
  dividend = dividend - divisor
  quotient = quotient + 1
}
return quotient
```

In this way, we can roughly evaluate the answer. However, there are many problem with this algorithm:

1. Cases limitation: This algorithm can't deal with cases where dividend is smaller than divisor, which will only return zero.
2. Precision : Also, the precision is not promised, for the quotient can only be added by 1 for each loop.
3. Time complexity : What's more, the time complexity is also large, just think of a case where we have very large dividend and very small divisor, then the times of loop will be (dividend / divisor).

So I check up with the divide algorithm that `bc` calculator used. Unluckily, I found that bc can only provide with an answer with only integer part, which is the same case above.

Finally, I improved the roughly thought algorithm with bit shifting, here is the pseudocode:

```
quotient = 0
offset = 0
while(dividend is bigger than a very small number)
{
    if(dividend is bigger or equal than divisor)
    {
      shift the divisor bitwise left until next shift will make it bigger than dividend
      after each shift, offset++
    }
    else//dividend is smaller than divisor
    {
      shift the divisor bitwise right until it just happend to be smaller than dividend
      after each shift, offset--
    }

    dividend = dividend - divisor * 10^(offset)
    quotient = quotient + 10^(offset)
}
```

```
return quotient
```

In this way, all problems of the previous algorithm are solved or at least improved:

1. The algorithm allows the dividend to be smaller than the divisor by shifting the divisor bitwise right
2. The precision can be very high limited by the "very small number" in the while loop
3. Because we did our best to make the divisor just smaller than the dividend by shifting bits, a large amount of time is reduced, assume the dividend is (a0 * 10^m), divisor is (b0 * 10^n), the very small number is (c0 * 10^p). Then the previous algorithm has time complexity of O(10^(m-n)) , and the optimized one has time complexity of O(10*(m-p) + (m-n+n-p) ) = O(m-p), which greatly improves the efficiency.

Detailed implementation please refer to `calculator.c`

**squre root**

Because the computer doesn't have a basic operation with square root, so we must find another way to implement it. To ensure the efficiency, I choose Newton's Iteration law to implement the function. Here is core block of the code:

```
do
{
    /* prev_x = x */
    strcpy(prev_x,x);

    /* Newton's iterative function : x = (x + a / x) / 2 */
    char two[] = "2";
    char * ans1 = divide(num,x);
    char * ans2 = add(ans1,x);
    char * ans3 = divide(ans2,two);
    strcpy(x,ans3);

    /* calculate the difference : delta = x - prev_x */
    strcpy(delta,subtract(x,prev_x,0));

    /* if(delta < 0) delta = -delta */
    if(delta[0] == '-')
    {
        for(int i = 1 ; i < strlen(delta) ; i++)
            delta[i-1] = delta[i];

        delta[strlen(delta)-1] = '\0';
    }

}while (strcmp(delta,epsilon) > 0);// stop iteration when difference is smaller than some very small number
```

Apart from the function's implementation , I found even when the precision is very high, some times the answer will still be not precise, like `sqrt(90000) = 299.999999999.....`, so , to supplement the answer to its correct value in such case, I add the last bit to generate exactly a carry and a zero left on the bit when the fraction part of the number becomes very long:

```
if(strlen(x) > 30)
{
    char * x_dot = strchr(x, '.');
    if(x_dot)
    {
        int x_Frac_len = strlen(x) - (x_dot - x);
        int last_digit = x[strlen(x)-1] - '0';
        if(last_digit > 0)
        {
            /* add the last bit to zero and add the carry to next bit */
        }
    }
}
```

**sin**

sin function is done based on Taylor expansion, whose  order can be modified freely. The whole processing procedure is also based on the add, subtract, multiply and divide functions.

```
/* convert the input into radian form */
char * radians = divide(multiply(a, PI), "180");

/* initialize */
char *x = malloc(OPERAND_MAX_SIZE * sizeof(char));
char *result = malloc(OPERAND_MAX_SIZE * sizeof(char));
char *factorial = malloc(OPERAND_MAX_SIZE * sizeof(char));
char *fact_n_str = malloc(OPERAND_MAX_SIZE * sizeof(char));

strcpy(x, radians);
strcpy(result, x);
strcpy(factorial, "1");

int fact_n = 2;
int sign = -1;

for (int i = 0; i < TAYLOR_EXPANSION_MAX_ORDER; i++)
{
    /* in each iterations, x = x * radian * radian */
    x = multiply(multiply(x, radians), radians);

    /* in each iterations, fact_value *= (fact_n+1) * (fact_n+2) */
    snprintf(fact_n_str, OPERAND_MAX_SIZE, "%d", fact_n++);
    factorial = multiply(factorial, fact_n_str);
    snprintf(fact_n_str, OPERAND_MAX_SIZE, "%d", fact_n++);
    factorial = multiply(factorial, fact_n_str);
```

```
    result = (sign == 1) ? add(result, divide(x, factorial)) : subtract(result, divide(x,
factorial), 0);
    sign *= -1;
}
```

However, as I started to input the numbers, when the number exceeds `140.345`, the answer suddenly grows to be a big minus number, and no precision adjustment is useful to get rid of this. This might be caused by big power of x or high order fractions. So, to make the answer more accurate, we need to project the value of parameter from the second, third and fourth quadrants into the first quadrant, where the number is smaller than 90:

```
while(subtract(a,"360",0)[0] != '-'){
    a=subtract(a,"360",0);
}

if(subtract(a,"90",0)[0] != '-' && subtract(a,"180",0)[0] == '-' )
    a=subtract("180",a,0);
else if(subtract(a,"180",0)[0] != '-' && subtract(a,"270",0)[0] == '-')
{
    if(strcmp(a,"180")!=0)
        *is_minus = 1;
    a=subtract(a,"180",0);
}
else if(subtract(a,"270",0)[0] != '-' && subtract(a,"360",0)[0] == '-')
{
    if(strcmp(a,"360")!=0)
        *is_minus = 1;
    a=subtract("360",a,0);
}
```

This way the answer will be accurate and calculate fast even when input number is very big.

**other functions**

There are other funtions in the code, but are easy to implement:

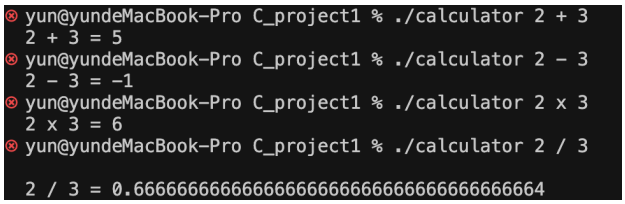
```
/* Some inner used functions */
double max(double a , double b);
double min(double a , double b);
char * trans_to_ans(int ans_Digits[],int dig_len,int decimal_pos);
char * tens_power_of(int power);

/* to output the answer in scientific notation */
int scientific_notation(char * answer, int ans_len, unsigned int precision,int max_e);
```

# Part III : Requirements Fullfill

The calculator meet the needs in the project document, andthe result is listed as follows:

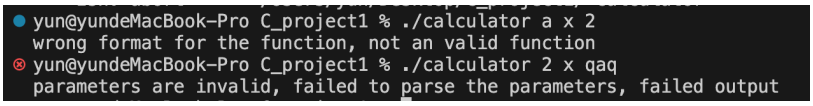
1. The implementation is done with C language and have only one file `calculator.c`


2. When you run the program as with arguments, it will print the expression and the result.

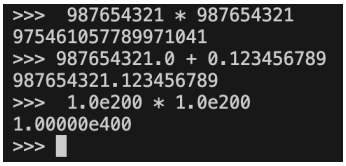



3. It can tell the reason why the operation cannot be carried out.

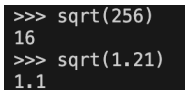

4. It can tell the reason when the input is not a number (In my implementation, if the first char is an alphabet, them the whole expression will be treated as a function)




5. If you input some big numbers, the output will still be valid( the scientific notation's precision can be modified)




6. Shown above
7. The calculator supports some frequently used functions:




# Part IV : AI Tools

The AI Tools I mainly used are `DeepSeek` and `Microsoft Copilot`. There are no different usage between them, and the reason I choose one instead of another is only based on if the server is busy for another one.

The usage of AI tools covers many aspects:

- Under circumstances where you have no Idea how to implement a function, just ask AI and it will provide with a list of choices, one of them will probably worth a try. For Example:

  *Q: How can I implement a sin function using c language without using librarys*

  The answer suggests Taylor expansion first, and specified its algorithm, implementation steps, example code in c, and the algorithms limit. Then , it also lists other ways such as 1.Lookup Table Approach 2.CORDIC Algorithm 3.Polynomial Approximations..... It seems to me that the Taylor expansion is a good choice, so I tried and did it.

- When you try to produce some code with high similarity but has detailed differences, AI tools are also useful. For example, I was about to implement `cos` function after I did `sin` functions, but the degree checking part is similar yet different in details, so I asked AI to *generate a cos version of this part of code shown as follows……*, and the output part runs well in logic.
- It is still hard for me to manage the storage when the source code grows big and complex, so after I'm done with one method, I will ask AI to help place the release the applied dynamic storage in the right place.

It is also worth reminding that AI generated codes can be a good reference or a guide, but the true implementation and detail dealing must be done manually, because the AI generated stuff is not promised to be correct in most times.