
Project 1 - Calculator

Name 颜云翔 (Yunxiang Yan)

SID 1912525

CS205 C/C++ PROGRAMMING

2022 FALL

PROJECT1 - CALCULATOR

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPT. OF COMPUTER SCIENCE AND ENGINEERING



Contents

1	Introduction	1
1.1	Challenges	1
1.2	Design Philosophy	1
2	Functionality	2
2.1	Input Checking	2
2.2	Valid inputs and huge numbers	3
3	External Libraries	4
3.1	Downloading	4
3.2	Installing	5
3.3	Install with install.sh	5
4	Implementation	6
4.1	Input checking	6
4.2	Computation methods	7
5	Conclusions	7

1 Introduction

The task of this project is to design and implement a simple calculator which can multiply two numbers in C++. In this section consists of the summary of the main challenges and the general philosophy I adopt in this project.

1.1 Challenges

This project task may seem easy at first glance but there are several non-trivial challenges that we need to tackle with consideration.

1. The first challenge lies in the detection of invalid inputs. In light of the arbitrary name of user inputs, we need to design a well-rounded validity detector that can find out these cases 1) the number of command line arguments is not 2 2) the inputs are not valid integers or floating-point number or numbers in scientific notation.

2. The second challenge is how to detect the number type (integer or floating-point number) and design appropriate methods for both situation so that the computation of integer multiplication will be exact while the computation of floating-point numbers will be as precise as possible.

3. The last and most critical challenge is how to handle the cases when the input numbers are extremely large (in some cases larger than any numeric datatype in C++).

1.2 Design Philosophy

This is an open project which can be completed in various ways. Thus, I listed the design philosophy behind my project implementation below:

1. Performance oriented. There are indefinite number of areas that we can possibly improve for this project but I chose to optimize the part related to performance (such as the biggest size of input that the program can take) rather than other minor details that may improve the visual effect (such as a GUI interface or more elegant formats).

2. Do not reinvent the wheel and try to stand on the shoulder of the giants. Even though implementing array-based bigint or bigfloat classes is very exciting, in order to follow the

first philosophy I chose to do a thorough research on the SOTA solutions for arbitrary precision arithmetic problem and used them in my project by configuring two open-source libraries: GMP and MPFR.

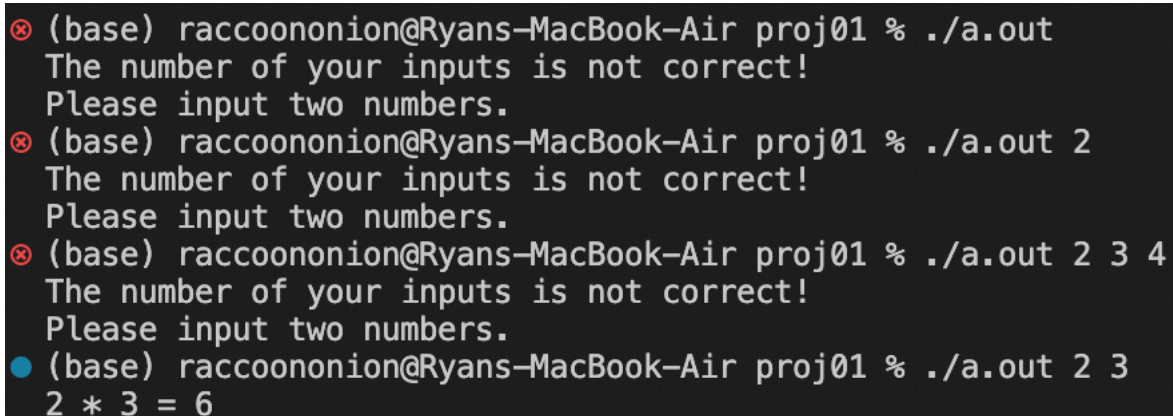
3. Readability matters. I try to make my code easy to read and understand by adding explanatory comments in the appropriate place and format the code according to Google C++ Style Guide. Besides, by separating the code into different functions, I try to avoid a too lengthy main function.

2 Functionality

The accepted valid inputs in this project are integers, floating-point numbers in digits form and numbers in scientific notation form. **There is no limit for the size of input integers other than all available memory of the computer that executes this program. For floating-point numbers and numbers in scientific notation, users can modify the precision as they wish so long as there is enough memory to assign.** In the rest of this section I will use examples to illustrate the input checking and large number computation.

2.1 Input Checking

If the number command line arguments is not 2, there will be error messages and the program will terminate. (Figure 1)

A terminal window screenshot showing four command-line interactions with a program named ./a.out. The first three commands result in error messages because the number of arguments is not 2. The fourth command succeeds and shows the output of a multiplication.

```
⊗ (base) raccoononion@Ryans-MacBook-Air proj01 % ./a.out
The number of your inputs is not correct!
Please input two numbers.
⊗ (base) raccoononion@Ryans-MacBook-Air proj01 % ./a.out 2
The number of your inputs is not correct!
Please input two numbers.
⊗ (base) raccoononion@Ryans-MacBook-Air proj01 % ./a.out 2 3 4
The number of your inputs is not correct!
Please input two numbers.
● (base) raccoononion@Ryans-MacBook-Air proj01 % ./a.out 2 3
2 * 3 = 6
```

Figure 1: Input number checks.

If the inputs cannot be interpreted as numbers in any conventional ways, an error message is generated and the program will terminate. I created a shell script for testing invalid inputs for simplicity. (Figure 2 & 3)

```
# invalid input format
echo "Inputs: a 1"
./a.out a 1
echo "Inputs: a1 1"
./a.out a1 1
echo "Inputs: 1a 1"
./a.out 1a 1
echo "Inputs: 1+1 1"
./a.out 1+1 1
echo "Inputs: 1-1 1"
./a.out 1-1 1
echo "Inputs: 1.0e1.2 1"
./a.out 1.0e1.2 1
echo "Inputs: 1.0e1e1 1"
./a.out 1.0e1e1 1
echo "Inputs: 1.1.1 1"
./a.out 1.1.1 1
echo "Inputs: 1. 1"
./a.out 1. 1
echo "Inputs: 1.e12 1"
./a.out 1.e12 1
echo "Inputs: e1 1"
./a.out e1 1
```

Figure 2: Shell scripts for invalid format input tests.

2.2 Valid inputs and huge numbers

Other than the formal ways of input, I also try to include some kinds of less formal but understandable input as valid. I tested the big integer multiplication and the big floating-point number multiplication cases presented in the project description PDF. I also tested some much bigger value. Figure (4 - 6)

```
❌ (base) raccoononion@Ryans-MacBook-Air proj01 % sh test.sh
Inputs: a 1
Your inputs cannot be interpret as numbers!
Inputs: a1 1
Your inputs cannot be interpret as numbers!
Inputs: 1a 1
Your inputs cannot be interpret as numbers!
Inputs: 1+1 1
Your inputs cannot be interpret as numbers!
Inputs: 1-1 1
Your inputs cannot be interpret as numbers!
Inputs: 1.0e1.2 1
Your inputs cannot be interpret as numbers!
Inputs: 1.0e1e1 1
Your inputs cannot be interpret as numbers!
Inputs: 1.1.1 1
Your inputs cannot be interpret as numbers!
Inputs: 1. 1
Your inputs cannot be interpret as numbers!
Inputs: 1.e12 1
Your inputs cannot be interpret as numbers!
Inputs: e1 1
Your inputs cannot be interpret as numbers!
```

Figure 3: Input format checks - invalid cases.

3 External Libraries

Since this project is based on the implementation of two open source projects: GNU MP and GNU MPFR . It is necessary to briefly introduce how to download and install the correct packages.

3.1 Downloading

Both libraries can be downloaded from their official websites: GNU MP: and MPFR . After the downloading process, it is recommended to verify the the security of the package by checking its DSA key. The detailed process and keys can be accessed in the downloading webpage as well.


```

bool detect_valid(const string &str) // By using const we avoid the possible distortion of original string
{
    long i = 0, j = str.length() - 1;

    // if string is of length 1 and the only
    // character is not a digit
    if (i == j && !(str[i] >= '0' && str[i] <= '9'))
        return 0;
}

```

Figure 8: One digit cases check.

4.2 Computation methods

We use `detect_int` to find out if the inputs are all integers. For all-integer case, we use GMP data types and functions. The result is accurate without a loss of precision. For cases with floating-point number(s), use MPFR data types and functions to get a result with 1024 bits precision and display the most significant 10 digits when outputting the result. Note that the precision as well as the number of significant digits printed can be modified by just change the two numbers in the code as long as enough memory can be assigned. However, in most cases, the default precision of 1024 bits and the default number of significant digits are enough.

5 Conclusions

During the entire process of completing this project, I get to know about the definition of arbitrary precision arithmetic and two SOTA libraries: GMP and MPFR. My biggest feelings after this project is the differences in open source environment between C++ and other languages like Java and python. Most of the time, there are no universal package managing tools and you have to download, install and configure the library by yourself. Besides that, I practiced a lot of knowledge learned in class and also get to educate myself on a lot of different areas by reading articles, documents online.

```

// To check if a '.' or 'e' is found in given
// string. We use this flag to make sure that
// either of them appear only once.
bool flag_e = false;
bool flag_dot = false;

for (; i <= j; i++) {
    // If any of the char does not belong to
    // {digit, +, -, ., e}
    if (str[i] != 'e' && str[i] != '.'
        && str[i] != '+' && str[i] != '-'
        && !(str[i] >= '0' && str[i] <= '9'))
        return 0;

    if (str[i] == '.') {
        // checks if the char 'e' or '.' has already
        // occurred before this '.' If yes, return 0.
        if (flag_e == true || flag_dot == true)
            return 0;

        // If '.' is the last character.
        if (i + 1 > str.length())
            return 0;

        // if '.' is not followed by a digit.
        if (!(str[i + 1] >= '0' && str[i + 1] <= '9'))
            return 0;

        // set flag_dot = 1 when . is encountered.
        flag_dot = true;
    }
}

```

Figure 9: Checking for dot symbol.

```

else if (str[i] == 'e') {
    // if e has appeared before
    if (flag_e)
        return 0;

    // if there is no digit before 'e'.
    if (!(str[i - 1] >= '0' && str[i - 1] <= '9'))
        return 0;

    // If 'e' is the last Character
    if (i + 1 == str.length())
        return 0;

    // if e is not followed either by
    // '+', '-' or a digit
    if (str[i + 1] != '+' && str[i + 1] != '-'
        && (str[i + 1] >= '0' && str[i] <= '9'))
        return 0;

    // set flag_e = 1 when e is encountered.
    flag_e = true;
}
else if (str[i] == '+' || str[i] == '-')
{
    if (str[i-1] != 'e')
        return 0;
}
}

// If the string skips all above cases, then it is numeric
return 1;
}

```

Figure 10: Checking for e symbol