CS205 C/ C++ Programming - Project 2 A Better Calculator

Name: Lv Yue
SID: 11710420

Part1: Analysis

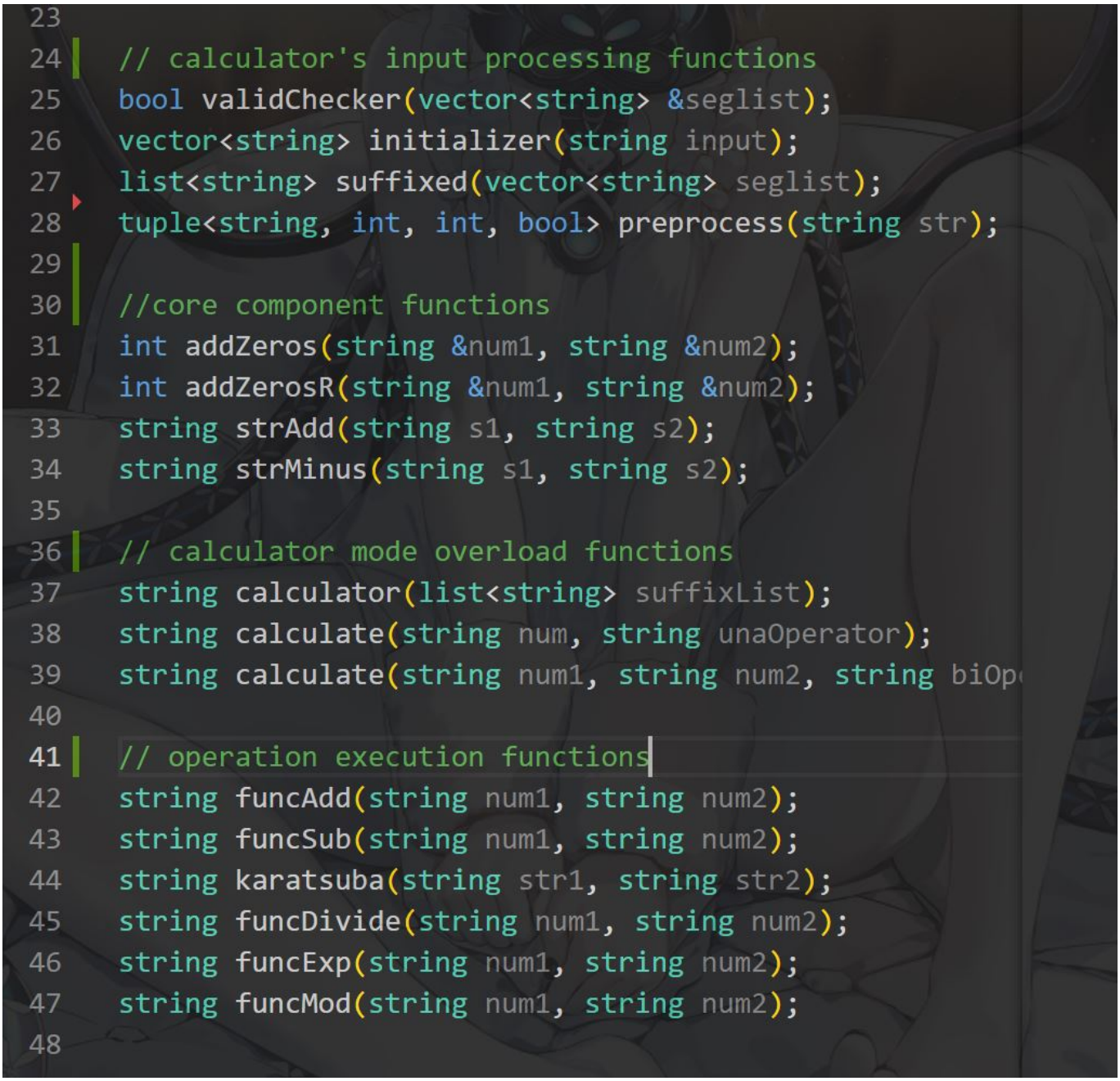
Project goal: Implement a much better calculator than that in Project 1.

It is envisaged that the calculator will support high precision addition + subtraction - multiplication * division / square root sqrt()->$ remainder % exponentiation ^ factorial ! operations.

In addition, the calculator supports user-defined variables.

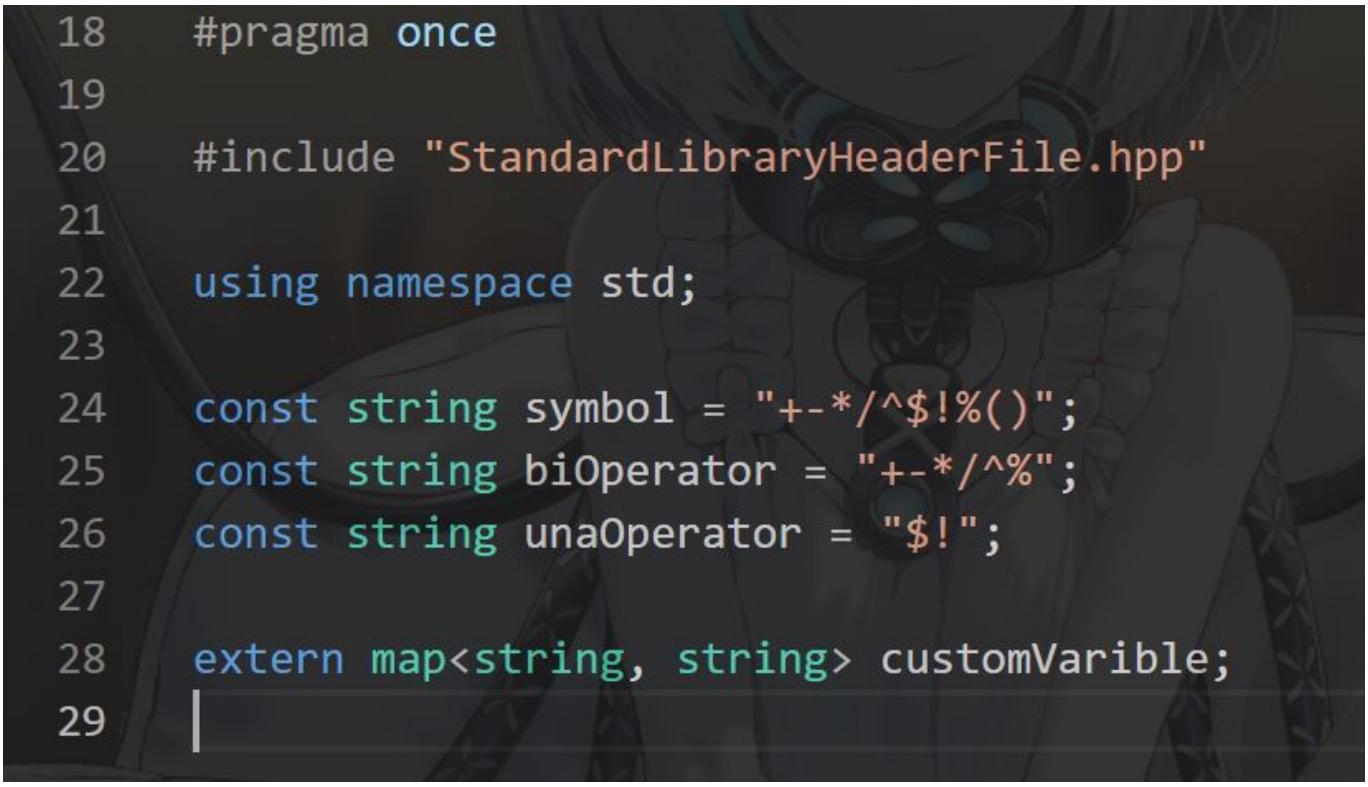
Part2: Core code

The **customFunction.hpp** declare the *calculator's input processing functions*, *core component functions*, *calculator mode overload functions* and *operation execution functions*:



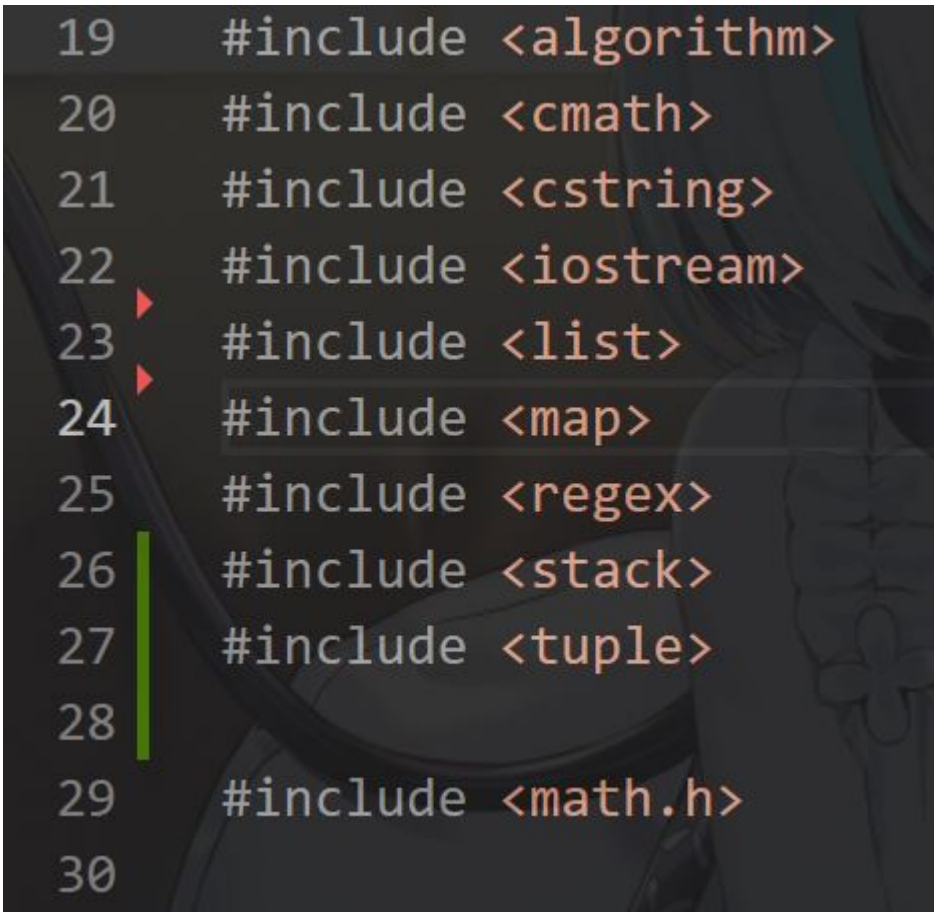
```
23
24  // calculator's input processing functions
25  bool validChecker(vector<string> &seglist);
26  vector<string> initializer(string input);
27  list<string> suffixed(vector<string> seglist);
28  tuple<string, int, int, bool> preprocess(string str);
29
30  //core component functions
31  int addZeros(string &num1, string &num2);
32  int addZerosR(string &num1, string &num2);
33  string strAdd(string s1, string s2);
34  string strMinus(string s1, string s2);
35
36  // calculator mode overload functions
37  string calculator(list<string> suffixList);
38  string calculate(string num, string unaOperator);
39  string calculate(string num1, string num2, string biOp
40
41  // operation execution functions
42  string funcAdd(string num1, string num2);
43  string funcSub(string num1, string num2);
44  string karatsuba(string str1, string str2);
45  string funcDivide(string num1, string num2);
46  string funcExp(string num1, string num2);
47  string funcMod(string num1, string num2);
48
```

The **globalVariables.hpp**, where a *map* is declared as *custom variable storage space* and multiple *operator string constants*:



```
18  #pragma once
19
20  #include "StandardLibraryHeaderFile.hpp"
21
22  using namespace std;
23
24  const string symbol = "+-*/^$!%()";
25  const string biOperator = "+-*/^%";
26  const string unaOperator = "$!";
27
28  extern map<string, string> customVarible;
29
```

The **StandardLibraryHeaderFile.hpp**, contains all required standard library header files:



```
19  #include <algorithm>
20  #include <cmath>
21  #include <cstring>
22  #include <iostream>
23  #include <list>
24  #include <map>
25  #include <regex>
26  #include <stack>
27  #include <tuple>
28
29  #include <math.h>
30
```

Finally the **headerCollections.hpp**, collect all the headers, to make .cpp files #include once and for all:

```
20  #include "StandardLibraryHeaderFile.hpp"
21
22  #include "globalVariables.hpp"
23
24  #include "customFunctions.hpp"
25
```

The project file has a ***brand new copyright header*** (sorry for the copyright header error caused by my previous oversight!):

src > calculator.cpp > ...

```
1   /*
2    Copyright (c) 2022 SUSTech - JustLittleFive
3
4    This program is free software: you can redistribute it and/or modify
5    it under the terms of the GNU General Public License as published by
6    the Free Software Foundation, either version 3 of the License, or
7    (at your option) any later version.
8
9    This program is distributed in the hope that it will be useful,
10   but WITHOUT ANY WARRANTY; without even the implied warranty of
11   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12   GNU General Public License for more details.
13
14   You should have received a copy of the GNU General Public License
15   along with this program.  If not, see <https://www.gnu.org/licenses/>.
16   */
```

(My GitHub home page: *https://github.com/JustLittleFive*)

The function implementation is divided into five project files:

1. **simpleCalculator.cpp**: Main function, as the program entry, handles the assignment statement of the custom variable.
2. **preprocess.cpp**: Preprocessing function implementation. Not only processing user input, but also preprocessing functions for scientific notation and decimal point are here.
3. **calculator.cpp**: Manages the order in which expressions are evaluated (suffix expression), and the overloaded calculate function handles unary and binary operators separately.
4. **calFunc.cpp**: The corresponding operation functions are implemented for all supported operators. Some temporarily unsupported inputs are also processed here.
5. **calComponents.cpp**: The common core component that implements string-type mathematical operations, and is used in almost every operator's implementation function.

... and the ***Karatsuba multiplier*** in **calFunc.cpp**:

```
/// @brief Multiply function by Karatsuba algorithm, inspired by the
article
/// https://www.geeksforgeeks.org/karatsuba-algorithm-for-fast-
multiplication-using-divide-and-conquer-algorithm/
/// @param str1
/// @param str2
/// @return string
string karatsuba(string num1, string num2) {
  ...

  int subLen = len / 2;
  string a = str1.substr(0, subLen);
  string b = str1.substr(subLen, len - subLen);
  string c = str2.substr(0, subLen);
  string d = str2.substr(subLen, len - subLen);

  string ac = karatsuba(a, c);
  string bd = karatsuba(b, d);
  // string acPbd = strAdd(ac, bd);
  // string adPbc = strAdd(karatsuba(a, d), karatsuba(b, c));
  string aPbcPd = karatsuba(strAdd(a, b), strAdd(c, d));
  string adPbc = strMinus(aPbcPd, strAdd(ac, bd));

  int bitShift = len - subLen;
  for (int i = 0; i < bitShift * 2; i++) {
    ac = ac + '0';
  }
  for (int i = 0; i < bitShift; i++) {
    adPbc = adPbc + '0';
  }

  result = strAdd(strAdd(ac, adPbc), bd);

  ...
}
```

There are many more annotations in the code too!

Part 3: Result & Verification

Some functions still lack support for precision,scientific notation or decimals.

Part 4 - Difficulties & Solutions

1. In order to ensure precision and no data loss, using string as an operator operation object requires rewriting the basic operation logic.
2. Code analysis somtimes shows that there are still riskes to be resolved: bunch of "Potential leak of memory", but I didn't use any `new` in my functions.
3. List in C++ is implemented in the form of linked list, which is different from other programming languages.
4. C++'s requirement that functions have only one return value forced me to use pointers and references.
5. Implementing support for scientific notation and decimals for various operators is more complicated than I thought.