CS205 C/ C++ Programming – Project 2

Name: 凌硕（Ling Shuo）
SID：11912409

Part 1 – Analysis

In this part, I want to describe my files and mark which subproblems they solved. And I will put the code that I think is important or highlighted in the second part.

File 1 – check.cpp

In this file, I created three functions to test whether a string can be understood as an integer, decimal, or scientific number.

File 2 – normalization.cpp

The function of this file is to transform an expression of a number to a standard scientific counting form. For example, $normalization(223e1) = 2.23e3$.

File 3 – mul.cpp

After the construction of before two files, we can start creating files to handle the operations of large numbers. I omit the functionality and the method of achieving multiplication in this file here since they're the same things in my first project.

Remark 1 : My idea is to firstly deal with the operation of positive integers, then change the decimal or scientific number into the form of the integer multiplied by several powers of ten, and do the integer operation already written, and finally deal with all numbers (including positive and negative numbers). It is worth mentioning that in the operations after, I also rely on such ideas to achieve, so the fifth subproblem has been achieved, that is, my program can operate on arbitrary precision of decimals or scientific numbers. As an example,

$$(-3.14e4) * (2e-3) = -[(314 * 10^2) * (2 * 10^{-3})] = -[(314 * 2) * (10^{2+(-3)})].$$

File 4&5 – add.cpp & sub.cpp

Firstly, I deal with the addition and subtraction of two positive integers (may be large numbers). In fact, the obvious way is to compute the every bits of the result respectively. When it comes to addition, suppose the two numbers are *a* and *b*. We store *a* in the array (of int) from the last bit forward, and do the same thing to *b*, the result is *a*[ ] and *b*[ ]. For example, if *a*=123, we can get the array *a*[3] with *a*[0]=3, *a*[1]=2 and *a*[2]=1. Suppose the sum of *a* and *b* is *c*, we construct a new array *c*[ ] which is wanted to store the each bits of the result. We

have the equation:

$$c[k] = a[k] + b[k].$$

Note that *c*[*k*] may large than 9 for some *k*, we need carry-over to ensure that each *c*[*k*] is a one-digit number. At last, we can output the result with the helping of *c*[*k*] (note that the order of *c*[*k*] and the order of the digits of the real result are inverse, that is, if the real result is 123, *c*[*k*] will be with *c*[*0*]=3, *c*[*1*]=2 and *c*[*2*]=1). The method of dealing with positive integer subtraction is completely like the process of pen-based subtraction, which concludes borrowing bits if necessary and getting the result one bit by one bit.

Secondly, I deal with additions and subtractions of decimals and scientific numbers as I said before. Suppose that we want to compute $a \pm b$ which a and b are decimals or scientific numbers. Then we want to transform them to $a = a_1 \times 10^{a_2}$ and $b = b_1 \times 10^{b_2}$ where a_1 and b_1 are integers. It is important that we should let $a_2 = b_2$ which is helpful to do addition or subtraction next. For instance, $2.3 - 5e - 1 = 23 \times 10^{-1} - 5 \times 10^{-1} = (23 - 5) \times 10^{-1}$.

Thirdly, to deal with all numbers which may be positive or negative, we can transform the equations such that we just need to compute the equations of two non-negative numbers. For example, if a and b are two positive numbers, then $a + (-b) = a - b;\ (-a) - (-b) = b - a$.

File 6 – div.cpp

It is not trivial to compute the division of two positive large numbers and I achieve it as the following steps. Firstly, computing the quotient and remainder of a division formula. Secondly, if the remainder is not zero, adding zeros to the end of the remainder and then compute it divides the original divisor and get a new quotient (how many zeros you add depends on the precision of the result you want, and in my code, the precision of my divisions is no less than four decimal places). Thirdly, the final result is obtained by scaling down the quotient obtained by the second time and adding the quotient obtained by the first time.

For example, if you want to compute $101 \div 3$, you firstly get the quotient 33 and remainder 2. Then we add five zeros at the end of the remainder, and compute $200000 \div 3 = 66666 \ldots 2$. Thus the result is $33 + 66666 \times 10^{-5} = 33.66666$. And after rounding, we are able to get the result accurate to four decimal places, that is 33.6667.

In order to deal with decimals and scientific numbers which can be both positive and negative, I finished the last work followed by the idea I mentioned in *remark1*.

File 7 – squarert.cpp

Finding the square root of a non-negative (large) integer is not an obvious problem. And I transform it to the problem of find the non-negative zero of simple functions. In this way, I finished the fourth subproblem.

In fact, computing the arithmetic square root of a non-negative integer a is equal to find the non-negative zero of the function $f(x) \triangleq x^2 - a$. So let's use the bisection method to roughly find the interval where the non-negative zero of f lies, and then we can use the Newton's iteration method to find the zero of f in the interval. (The detailed code is shown in Part 2.)

Remark 2: More specifically, we select an arbitrary x_0 in the interval and define x_n by $x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, n \geq 1$. The Newton's iteration method tells us that $x_n \to \sqrt{a}$ as $n \to +\infty$. (In fact, x_n converges to $\sqrt{a}$ at least quadratically, which is faster than the bisection method.)

File 8 – power.cpp

In order to enrich the content of the fourth subproblem, I wrote a file to calculate the integer power of an arbitrary nonnegative integer which is based on the previously written file for large number multiplication and the recursive method. To compute a^b, we just need to handle the case that $b > 0$, since $a^{-b} = 1 \div a^b$. Now, if $b = 2k$ for some $k \in \mathrm{Z}_+$, $a^b = (a \times a)^k$. If $b = 2k + 1$ for some $k \in \mathrm{Z}_+$, $a^b = a \times (a \times a)^k$. That is the reason why we can use the recursive method to compute a^b based on large number multiplication. (The detailed code is shown in Part 2.)

File 9 – main.cpp

With the above files which can achieve large number operations, we can start working on the first few subproblems.

In order to achieve the function of the first subproblem, we can use the recursive method. Suppose we construct a function `calculator1` to achieve it. Specifically, we can firstly find the sign "*" and "/", and do these operator (do the first multiplication or division from left firstly). And then we apply this function `calculator1` to the rest of the string and the result we just got. If there is no sign "*" and "/", we handle the sign "+" and "-" as the same method. Returning the string until there is no sign for the four operations (except for the minus sign at the begin of a string to represent a negative number). So we can use recursive method to achieve it. (The detailed code is shown in *main.cpp*, please see the function `calculator1`).

In order to achieve the function of the second subproblem, we can also use the recursive method. Suppose we construct a function `calculator` to achieve it. Now we should find "(" and ")" firstly and do the operation (by the function `calculator1` we have finished) between one pair of "(" and ")" if there is no more parentheses between the two parentheses. And then handle the remainder string combining with the result of the operation between the two parentheses by function `calculator`. Until there are no parentheses (or parentheses only exist at the beginning and end of string), we can apply

the function `calculator1` to the string and return the result. (The detailed code is shown in *main.cpp*, please see the function `calculator`).
In order to achieve the function of the third subproblem, we can use *cin.getline* and loop structure (combing with appropriate termination conditions: no sign "=" in a line) to achieve input and replace the variables in the equation by their values to get the result. (The detailed code is shown in Part 2, please see line 48 to 112 in main.cpp).

File 10 – f.h

A common head file.

File 11 – CMakeLists.txt

To manage files and get an executable file called "*calculator*", there is content in the CMake file:

```
cmake_minimum_required(VERSION 3.10)
project(calculator)
aux_source_directory(. DIR_SRCS)
add_executable(calculator ${DIR_SRCS})
```

Part 2 – Code

In this part, I put the code that I think is important or highlighted.

1 – squarert.cpp

```
#include <iostream>
#include <string>
#include "f.h"

using namespace std;

string square_root(string a,string times)  //compute the aquare root of a positive integers
{
    string step="1e"+std::to_string(a.size()-1);
    if(isNum(times)==false||isNum(a)==false)
    {
        return "Sorry! This is illegal input!";
    }
    else
    {
        string result;
```

```
        //a^0.5 is the positive zero of the function f(x)=x^2-a, then we can gain this by the
Newton's iteration method
        string flag="0";
        while(true)
        {
            string a1=sub(mul(flag,flag),a);
            string a2=sub(mul(add(flag,step),add(flag,step)),a);
            if(sub(a1,"0")=="0")
            {
                result=flag;
                break;
            }
            else if(sub(a2,"0")=="0")
            {
                result=add(flag,step);
                break;
            }
            else
            {
                if((int)sub(a1,"0")[0]==45&&(int)sub(a2,"0")[0]!=45)  //a1<0&&a2>0
                {
                    result=newton_iteration(a,flag,add(flag,step),std::stoi(times));
                    break;
                }
                else
                {
                    flag=add(flag,step);
                }
            }
        }

        return result;

    }


}

string newton_iteration(string a,string a1,string a2,int times)  //find the zero of f(x)=x^2-a
within [a1,a2]
{
    string p0=div(add(a1,a2),"2");  //initial point
    int N=times;  //iteration times
    string p;
```

```
    for(int i=0;i<N;i++)
    {
        p=sub(p0,div(sub(mul(p0,p0),a),mul("2",p0)));
        p0=p;
    }
    return p;
}
```

Remark 3: To manage the non-negative number smaller than 10^20, we recommend between 20 and 40 iteration times. The larger number need more times of iteration which will greatly extend the operation time.

2 – power.cpp

```
#include <iostream>
#include <string>
#include "f.h"

using namespace std;

string power(string a, string b)  //a>=0, a and b are both integers
{//if you want to conpute the case which a is negative, you just need to tell the sign of abs(a)^b
which can gained by my code
    int sign_b=1;
    if((int)b[0]==45)  //"-"
    {
        b.erase(0, 1);
        sign_b=-1;
    }
    if(isNum(a)==0||isNum(b)==0)
    {
        return "Sorry! This is illegal input!";
    }
    else if(sub(a,"0")=="0"&&sub(b,"0")=="0")
    {
        return "Sorry! This is illegal input!";
    }
    else if(sub(a,"0")!="0"&&sub(b,"0")=="0")
    {
        return "1";
    }
    else if(sub(a,"0")=="0"&&sub(b,"0")!="0")
    {
        return "0";
```

```
    }
    else if(sign_b==1)
    {
        return power_compute(a,b);
    }
    else
    {
        return div("1",power_compute(a,b));
    }
}

string power_compute(string a,string b)  //here a and b are both positive integers
{
    string result="1";
    if(b.size()<=2)
    {
        for(int i=0;i<std::stoi(b);i++)
        {
            result=mul_positive_integers(result,a);
        }
    }
    else
    {
        if(ddiv_positive_integers(b,"2").remainder=="0")
        {
            result=power_compute(mul_positive_integers(a,a),ddiv_positive_integers(b,"2").quotient);
        }
        else
        {
            result=mul_positive_integers(a,power_compute(mul_positive_integers(a,a),ddiv_positive_int
egers(b,"2").quotient));
        }
    }
    return result;
}
```

3 – main function in main.cpp

```
#include <iostream>
#include <string>
#include <cstring>
#include "f.h"
using namespace std;
```

```
string calculator1(string str);  //without parentheses
string calculator(string str);  //allow parentheses

int main()
{
    cout<<"Please select the model you want!"<<endl;
    cout<<"1. Four fundamental operations;"<<endl;
    cout<<"2. Four fundamental operations with variables you can define;"<<endl;
    cout<<"3. Find the square root of a positive integer;"<<endl;
    cout<<"4. Compute the integer power of a positive integer."<<endl;

    string m;
    cin>>m;
    if(m=="1")
    {
        cout<<"Please enter the expression that you want to compute:"<<endl;
        cout<<"Notes: 1. Please use '+', '-', '*', '/' to express the sign of four fundamental operations;"<<endl;
        cout<<"       2. Please do not enter spaces;"<<endl;
        cout<<"       3. You can enter integers, decimals and numbers given by the scientific counting
method;"<<endl;
        cout<<"       4. You can enter numbers which are beyond the range of int, or even of long int;"<<endl;
        cout<<"       5. In division, the result is accurate to four decimal places."<<endl;
        string a;
        cin>>a;
        int flag=0;
        for(int i=0;i<a.size();i++)
        {
            if((((int)a[i]>=40&&(int)a[i]<=43)||((int)a[i]>=45&&(int)a[i]<=57)||(int)a[i]==69||(int)a[i]==101)==fals
e)
            {
                flag=1;
                break;
            }
        }
        if(flag==1)
        {
            cout<<"Sorry! This is illegal input!"<<endl;
        }
        else
        {
            cout<<"The result is: "<<calculator(a)<<endl;
        }
    }
    else if(m=="2")
```

```
    {
        cout<<"Warning: 1. You cannot use 'e' or 'E' to define you variables since they are used in the scientific
counting method;"<<endl;
        cout<<"         2. Your variables should have only one length;"<<endl;
        cout<<"         3. Numbers, '+', '-', '*', '/', '.','(', ')' are not allowed to be contained in the names of
the variables."<<endl;
        cout<<"         4. Please do not enter spaces in the equations that define variables"<<endl;
        string b;
        char* s = new char[1024];

        cin.ignore();
        while (cin.getline(s, 1024))
        {
            int flag=0;
            for(int i=0;i<strlen(s);i++)
            {
                if((int)s[i]==61)  //=
                {
                    flag=1;
                    break;
                }
            }
            b += s;
            b += '\n';
            if (flag == 0)
            {
                break;
            }
        }
        delete[]s;

        int num_of_v=-1;
        for(int i=0;i<b.size();i++)
        {
            if(b[i]=='\n')
            {
                num_of_v=num_of_v+1;
            }
        }
        string variables[num_of_v];
        string equation;
        for(int i=0;i<num_of_v;i++)
        {
            variables[i]=b.substr(0,b.find('\n'));
```

```
            b.erase(0,b.find('\n')+1);
        }
        equation=b.substr(0,b.size()-1);

        string library="1234567890.+-*/()Ee";
        for(int i=0;i<equation.size();i++)
        {
            if(library.find(equation[i])==library.npos)
            {
                for(int j=0;j<num_of_v;j++)
                {
                    if((int)variables[j][0]==(int)equation[i])
                    {
                        equation=equation.replace(i,1,variables[j].substr(variables[j].find("=")+1,variables[j].size(
))));

                        break;
                    }
                }
            }
        }

        cout<<"The result is: "<<calculator(equation)<<endl;
    }
    else if(m=="3")
    {
        cout<<"Please enter the radicand:"<<endl;
        string c1;
        cin>>c1;
        cout<<"Please enter the times of iterations you want:"<<endl;
        cout<<"(To manage the number smaller than 10^20, we recommend between 20 and 40 times. The larger number
need more times of iteration which will greatly extend the operation time!)"<<endl;
        string c2;
        cin>>c2;
        cout<<"The result is: "<<square_root(c1,c2)<<endl;
        //The operations for determining whether c1 and c2 are valid have been written in squarert.cpp.
    }
    else if(m=="4")
    {
        cout<<"Please enter the base number: (you can enter an arbitrary non-negative integers)"<<endl;
        string d1;
        cin>>d1;
        cout<<"Please enter the exponent: (you can enter an arbitrary integers)"<<endl;
        string d2;
        cin>>d2;
```

```
        cout<<"The result is: "<<power(d1,d2)<<endl;
        //The operations for determining whether d1 and d2 are valid have been written in power.cpp.
    }
    else
    {
        cout<<"Sorry! This is illegal input!"<<endl;
    }

}
```

Part 3 - Result & Verification

Subproblem1:

```
shuo_lin@LAPTOP-CM9SC1IR:~/project2$ ./calculator
Please select the model you want!
1. Four fundamental operations;
2. Four fundamental operations with variables you can define;
3. Find the square root of a positive integer;
4. Compute the integer power of a positive integer.
1
Please enter the expression that you want to compute:
Notes: 1. Please use '+', '-', '*', '/' to express the sign of four fundamental operations;
       2. Please do not enter spaces;
       3. You can enter integers, decimals and numbers given by the scientific counting method;
       4. You can enter numbers which are beyond the range of int, or even of long int;
       5. In division, the result is accurate to four decimal places.
5+2*3
The result is: 1.1E1
```

Subprobelm2:

Case1:

```
(5+2)*3
The result is: 2.1E1
```

Case2: (more difficult)

```
(2312832+21312-212)/((2312+3232)*(1.1+9e-1))
The result is: 2.10491703E2
```

Subproblem3:

```
shuo_lin@LAPTOP-CM9SC1IR:~/project2$ ./calculator
Please select the model you want!
1. Four fundamental operations;
2. Four fundamental operations with variables you can define;
3. Find the square root of a positive integer;
4. Compute the integer power of a positive integer.
2
Warning: 1. You cannot use 'e' or 'E' to define you variables since they are used in the scientific counting method;
         2. Your variables should have only one length;
         3. Numbers, '+', '-', '*', '/', '.','(', ')' are not allowed to be contained in the names of the variables.
         4. Please do not enter spaces in the equations that define variables
x=3
y=6
x+2*y
The result is: 1.5E1
```

Subproblem4:

Case1:

```
shuo_lin@LAPTOP-CM9SC1IR:~/project2$ ./calculator
Please select the model you want!
1. Four fundamental operations;
2. Four fundamental operations with variables you can define;
3. Find the square root of a positive integer;
4. Compute the integer power of a positive integer.
3
Please enter the radicand:
3
Please enter the times of iterations you want:
(To manage the number smaller than 10^20, we recommend between 20 and 40 times.
5
The result is: 1.732050807568877293527446 34E0
```

Case2: (More difficult)

```
Please enter the radicand:
1234567890987654321
Please enter the times of iterations you want:
(To manage the number smaller than 10^20, we recommend between 20 and 40 times. The larger number need more times of iteration which will greatly extend the operation time!)
40
The result is: 1.11111110649999990437499603156247941612097417353656042693426325211251505937501 24E9
```

Part 4 - Difficulties & Solutions

1. Finding the square root of a non-negative (large) integer is not trivial. I transform it to the problem of find the non-negative zero of simple functions and I use Newton's iteration method to solve it. In fact, this kind of method can be used to compute any times root of a positive integer.
2. Exponentiation using a recursive method can speed up the operation.
3. Using *cin.getline* and loop structure (combing with appropriate termination conditions) to solve the variable input problem.
4. Giving the user four functional choices and appropriate operation tips in my code.