浮点数的表示与算术运算算法

姓名：应承峻

学号：3170103456

实验描述

浮点数的表示与算术运算算法分析，要求理论推导与程序模拟。 算术运算包括字符串到浮点数、浮点数到字符串的转换，加、减、乘、除四则运算等。要有说明文档，包括算法证明、程序框图、使用方法、特殊处理(溢出、数位扩展)、实例分析等等。字符串转换可能稍难。

```
typedef unsigned int dwrd;
char* ftoa(dwrd);
dwrd atof(char*);
dwrd fadd(dwrd, dwrd);
dwrd fsub(dwrd, dwrd);
dwrd fmul(dwrd, dwrd);
dwrd fdiv(dwrd, dwrd);
```

- 主程序只供验证，你所须写的就是6个子程序。`atof`和`ftoa`可能会比较麻烦。
- 对结果应该进行分析，讨论你的浮点运算适用范围，可能的问题等等。
- 验证要有大数、小数，比如：`12345678901234567890123456789.123456789`。浮点的范围是十的正负38次方。如果不能，分析原因。
- 可以增加十六进制的对比。比如显示`z1.f`与`(u.f+v.f)`的十六进制。这样可以分析四舍五入情况。

算法分析

加减法

浮点数加法的原则为：“小阶向大阶看齐“，根据浮点数的格式取出ex, ey分别为浮点数x和y的阶码，而mx, my为其尾数。计算时，将小阶对应的尾数右移后向大阶看齐，然后将尾数相减得到新的尾数。相减后进行重新规格化，但此时要注意如果两尾数相减后为0则直接返回0即可否则会导致死循环。

浮点数的减法可以通过加法来实现：$x - y = x + (-y)$，因此只需改变y的符号位后再与x做加法即可。

```
dwrd fadd(dwrd x , dwrd y) {
    if (x == 0) return y;
    else if (y == 0) return x;
    if ((x << 1) < (y << 1)) return fadd(y , x); // |x| >= |y|
    dwrd ex = (x >> 23) & 0xFF;
    dwrd ey = (y >> 23) & 0xFF;
    dwrd mx = (x & 0x7FFFFF) | 0x800000;
    dwrd my = (y & 0x7FFFFF) | 0x800000;
```

```
    my >>= (ex - ey);
    if ((x^y) & 0x80000000) {  //case -
        mx -= my;
        if (mx == 0) return 0;  //zero
        while ((mx & 0x00800000) == 0) {
            mx <<= 1;
            ex--;
        }
    } else {  //case +
        mx += my;
        if (mx == 0) return 0; //zero
        while (mx & 0xFF000000) {
            mx >>= 1;
            ex++;
        }
    }
    return (x & 0x80000000) | (ex << 23) | (mx & 0x7FFFFF);
}

dwrd fsub(dwrd x , dwrd y) {
    return fadd(x , y ^ 0x80000000);
}
```

乘法

乘法的阶码运算法则，设$E(x)$为阶码，x为阶：

$E(x) = x + M$

$E(xy) = x + y + M = E(x) + E(y) - M$

根据真值表，浮点数乘法的符号位即为若两数最高位异或值：

x	y	$(x \oplus y) \wedge 0x80000000$
1	0	1
1	1	0
0	1	1
0	0	0

乘法的尾数即为两浮点数尾数值相乘，但由于一个浮点数的尾数（此处加上最前面的1）有24位，两个浮点数相乘时达48位，导致发生溢出。所以，程序中只取两个尾数相乘后的前24位作为新的尾数而舍弃了后24位。

浮点数乘法后可能会产生类似加法一样的进位问题，所以在得到结果后也要进行规格化处理 。

```
dwrd fmul(dwrd x, dwrd y) {
    if (x == 0 || y == 0) return 0;
    dwrd ex = (x >> 23) & 0xFF;
    dwrd ey = (y >> 23) & 0xFF;
    dwrd mx = (x & 0x7FFFFF) | 0x800000;  //24-bit
    dwrd my = (y & 0x7FFFFF) | 0x800000;
```

```
    dwrd mr = 0;
    ex = ex + ey - 127;
    while (my != 0) {   //乘法算法
        mr >>= 1;
        if (my & 1) mr += mx;
        my >>= 1;
    }
    while (mr & 0xFF000000) {  //规格化
        mr >>= 1;
        ex++;
    }
    return ((x^y) & 0x80000000) | (ex << 23) | (mr & 0x7FFFFF);
}
```

除法

除法的阶码运算法则，设$E(x)$为阶码，x为阶：

$E(x/y)=x-y+M=E(x)-E(y)+M$

尾数除法采用的算法为恢复除数的除法，为了便于对算法的理解我们以$4\div10$为例进行演示

4.0在浮点数中表示为：0 10000001 00000000000000000000000

10.0在浮点数中表示为：0 10000010 01000000000000000000000

首先通过阶码的运算法则得到阶码为$10000001-10000010+01111111=01111110\,(126)$

然后对尾数做除法$10000000000000000000000\div10100000000000000000000$

为了处理最高位商0导致的移位失败，我们在开始时通过不断的移位处理使得$mx>my$，这样保证了最高位能够商1，除此之外，该操作也能够保证除法后不需要再进行借位规格化处理。此外，该操作也能排除左移导致的进位规格化处理的问题，证明如下：设最后一次移位前$x/y=\alpha$,由移位条件得$\alpha\in[1/2,1)$，当再进行移位后，可以得到$\alpha_1\in[1,2)$，因此商必介于1与2之间，不会产生进位。

每次除法时，如果$mx<my$，则说明不够减，此时商0，否则将mx减去my并且商1，最后每次减法后将被除数左移，直到商的第24位为1为止。

```
dwrd fdiv(dwrd x , dwrd y) {
    if (x == 0) return 0;
    if (y == 0) {
        cout << "除数不能为0! " << endl;
        return (x & 0x80000000) | 0x7F800000; //INF
    }
    dwrd ex = (x >> 23) & 0xFF;
    dwrd ey = (y >> 23) & 0xFF;
    dwrd mx = (x & 0x7FFFFF) | 0x800000;
    dwrd my = (y & 0x7FFFFF) | 0x800000;
    dwrd mr = 0;
    ex = ex - ey + 127;
    while (mx < my) {  //handle borrow before calculate
        mx <<= 1;
        ex--;
    }
    while ((mr & 0xFF800000)==0) {
```

```
        if (mx < my) mr <<= 1;  //quotient 0
        else {
            mx -= my;
            mr = (mr << 1) | 1; //quotient 1
        }
        mx <<= 1;
    }
    return ((x^y) & 0x80000000) | (ex << 23) | (mr & 0x7FFFFF);
}
```

浮点数转换成字符串

浮点数转换成字符串较为繁琐，首先需要考虑特殊情况，如果是0则直接输出0，如果阶码是255则直接输出INF表示无穷大。其次是考虑符号位，根据最高位判断出符号位后直接加到字符串中。

接来需要分情况讨论：x代表一个大于1的数还是小于1的数。

如果是一个大于1的数，则需要得到其整数部分和小数部分。

以$12(1100.00000000000000000000)$为例

整数部分较为容易，用高精度算法直接乘幂即可。而小数部分的处理比较麻烦。由于$x \div 2 = x \times 5/10$，因此我们可以通过"补0乘5"的方式得到每次运算的权重，如$0.5 - 0.25 - 0.125 - 0.0625$的权重变化可以看做是：

$5 -> 05 * 5 = 25 -> 025 * 5 = 125 -> 0125 * 5 = 0625 -> ...$这一变换得到的，因此我们每次构造一个以5位幂的权重进行计算即可。

如果x是一个小于1的数，则不需要处理整数部分，小数部分的处理方法同上。

```
string ftoa(dwrd x) {
    if (x == 0) return "0";  //special case zero
    string buffer;
    if (x & 0x80000000) buffer += '-';  //x<0
    dwrd ex = (x >> 23) & 0xFF;   //exp
    dwrd mx = (x & 0x7FFFFF) | 0x800000; //mx
    dwrd integer = 0 , decimal = 0;
    if (ex == 255) buffer += "INF";  //infinity
    else if (ex >= 127) {
        ex -= 127;
        string result = "0" , power = "1";
        for (int k = ex + 1; k > 0; k--) {
            if ((mx >> (24 - k)) & 1) result = add(result , power);
            power = multi(power , 2);
        }
        buffer = buffer + result + ".";
        result = "0";  power = "5";
        for (int k = ex + 2; k <= 24; k++) {
            if ((mx >> (24 - k)) & 1) {
                string temp = power;
                reverse(temp.begin() , temp.end());
                result = add(result , temp);
            }
            power = "0" + power;
            power = multi(power , 5);
```

```
        }
        reverse(result.begin() , result.end());
        buffer += result;
    } else {  //<1
        buffer += "0.";
        int len = 24;
        string result="0" , power = "5";
        for (dwrd k = 1; k < 127 - ex; k++) {
            power = "0" + power;
            power = multi(power , 5);
        }
        while ((mx & 1) == 0) {
            mx >>= 1;
            len--;
        }
        while (len > 0) {
            if ((mx >> (len - 1)) & 1) {
                string temp = power;
                reverse(temp.begin() , temp.end());
                result = add(result , temp);
            }
            power = "0" + power;
            power = multi(power , 5);
            len--;
        }
        reverse(result.begin() , result.end());
        buffer += result;
    }
    return buffer;
}
```

字符串转换成浮点数

在将字符串转换成浮点数前，首先通过正则表达式检验给定字符串是否是一个合法的浮点数。

为了能够分别计算整数部分和小数部分，我们将整个字符串拆分成三部分：符号位、整数子串和小数子串。拆分后，需要考虑一个特殊情况：即浮点数为0的情况，此时应直接返回。

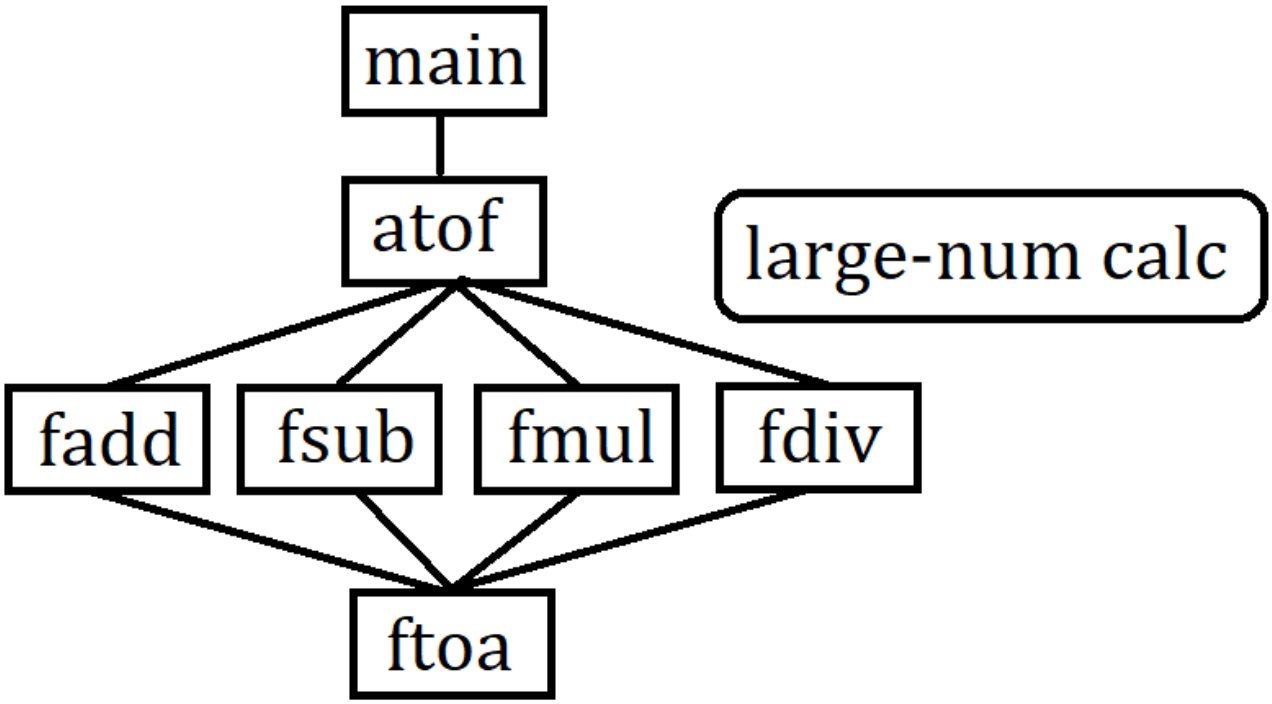
整数部分的转化思路为除二取余，以此得到整数部分的二进制串。小数部分的转换比较麻烦，我们以0.1234为例子，通过字符串切割得到小数部分1234，乘二取整的结果取决于是否前面多出了一位，比如5001乘二时得到了10002，此时需要取证，而$1234 \times 2 = 2468 < 10000$不需要取证，因此设小数部分的字符串长度为`dotsize`，则乘二取整得到`1`的条件为`multi(decimal,2).size()>=dotsize`。

将整数和小数的字符串拼接后截取前23位作为尾数，再与符号位和阶码拼接即可。

```
dwrd atof(string s) {
    bool borrow = false;
    bool check = regex_match(s , regex("^[-]?[0-9]*\.?[0-9]+$"));
    if (!check) {
        cout << "不是一个合法的浮点数" << endl;
        return 0;
    }
    string bin , integer = "0" , decimal = "0" , res;
```

```
    dwrd pos , i = 0;
    dwrd ex = 0 , x = 0;
    if (s.at(0) == '-') { //calc symbol
        res += '1';
        i++;
    } else res += '0';
    res += "00000000";
    pos = i;
    while (pos != s.size() && s.at(pos) != '.' ) pos++;  //seperate
    if (pos < s.size()) decimal = s.substr(pos + 1);
    integer = s.substr(i , pos - i);
    if (!integer.compare("0") && !decimal.compare("0")) return 0;  //special case for input zero
    if (integer.compare("0") == 0) bin = "0";
    else {
        while (integer.compare("0") > 0) { //convert to binary
            bin += (mod(integer , 2)) ? "1" : "0";
            integer = divide(integer , 2);
        }
        reverse(bin.begin() , bin.end());
    }
    ex = bin.size() - 1 + 127;   //evaluate ex
    dwrd dotsize = decimal.size();
    while (bin.size() < 46) {   //construct 23-bit format mx
        decimal = multi(decimal , 2);
        if (decimal.size()>dotsize) {
            decimal.erase(decimal.begin());
            bin += "1";
        } else bin += "0";
    }
    while (bin.at(0) == '0') {   //case for type 0.***
        bin.erase(bin.begin());
        ex--;
    }
    bin.erase(bin.begin());
    res += bin.substr(0 , 23);
    for (i = 0; i < 32; i++) x = x * 2 + (res.at(i) - '0');
    return x + (ex << 23);
}
```

程序框图



特殊处理

针对`unsigned int`在字符串与浮点数相互转换的溢出情况，我们采用了高精度算法进行计算使其能够达到数位扩展的要求，其算法如下。

```
string divide(string s , dwrd b) {  //large-number divede algorithm
    dwrd r = 0;
    dwrd i = 0;
    char c[100];
    char *p;
    string res;
    for (i = 0; i < s.size(); i++) {
        r = r * 10 + s[i] - '0';
        if (r < b) c[i] = '0';
        else {
            c[i] = r / b + '0';
            r = r % b;
        }
    }
    c[i] = '\0';
    p = c;
    while ((*p == '0') && p) p++;
    return p;
}

dwrd mod(string s , dwrd b) {  //large-number mod algorithm
    dwrd r = 0;
    dwrd i = 0;
    char c[100];
    char *p;
```

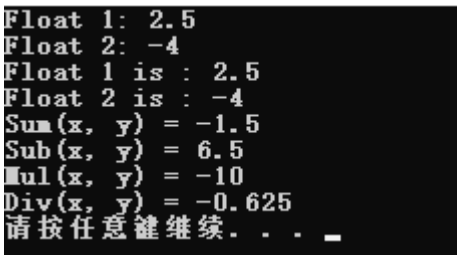
```
    string res;
    for (i = 0; i < s.size(); i++) {
        r = r * 10 + s[i] - '0';
        if (r < b) c[i] = '0';
        else {
            c[i] = r / b + '0';
            r = r % b;
        }
    }
    c[i] = '\0';
    p = c;
    while ((*p == '0') && p) p++;
    return r;
}


string multi(string s , dwrd b) { //large-number multiple algorithm
    char c[100];
    dwrd len = 0;
    dwrd carry = 0;
    for (int i = s.size() - 1; i >= 0; i--) {
        dwrd temp = (s[i] - '0') *b + carry;
        c[len++] = temp % 10 + '0';
        carry = temp / 10;
    }
    while (carry != 0) {
        c[len++] = carry % 10 + '0';
        carry /= 10;
    }
    c[len] = '\0';
    string res = c;
    reverse(res.begin() , res.end());
    return res;
}


string add(string x , string y) { //large-number add algorithm
    char c[100];
    string res;
    dwrd len = 0;
    dwrd carry = 0;
    reverse(x.begin() , x.end());
    reverse(y.begin() , y.end());
    while (x.size() < y.size()) x += '0';
    while (y.size() < x.size()) y += '0';
    for (int i = 0; i < x.size(); i++) {
        int temp = x[i] - '0' + y[i] - '0' + carry;
        c[len++] = temp % 10 + '0';
        carry = temp / 10;
    }
    if (carry != 0) {
        c[len++] = carry + '0';
    }
```

```
    c[len] = '\0';
    res = c;
    reverse(res.begin() , res.end());
    return res;
}
```

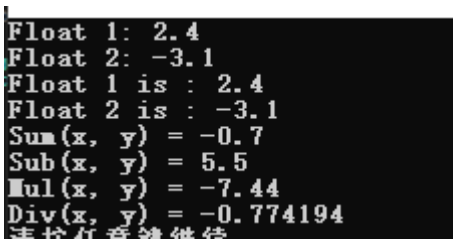
使用方法与实例分析

使用方法：根据提示输入两个浮点数即可。

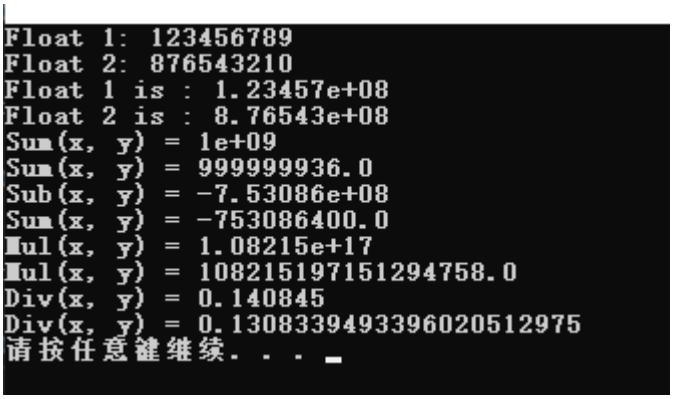
测试样例1：能够转换成二进制的浮点数，带符号



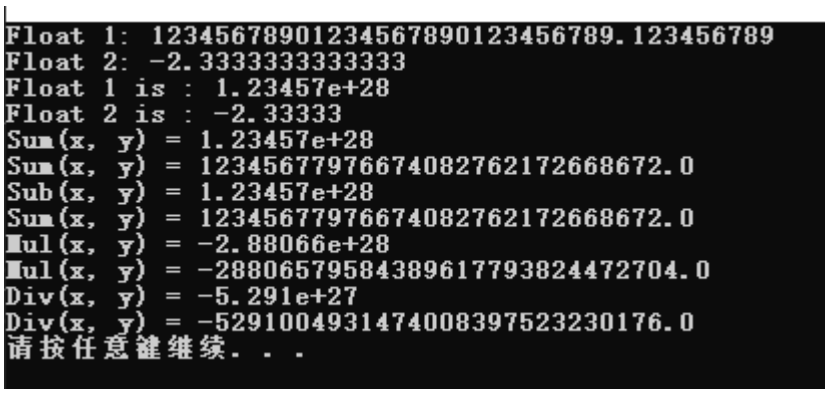
测试样例2：不能够准确转换成二进制的浮点数，带符号



测试样例3：大数运算（分两行，前面一行为标准输出，后一行为atof函数输出）



测试样例4：大数运算（分两行，前面一行为标准输出，后一行为atof函数输出）



分析：本程序浮点数适用范围：由于采用了高精度算法，因此在10的正负38次方内均可以使用。但是由于`atof`函数在转换输出时存在一定的误差，因此针对不能完全转换成二进制浮点数的数的输出会存在精度丢失的问题。