浮点数加减运算程序原理说明

（程序运行测试结果在最后）

根据题目的要求;首先需要自定义一个浮点类型。本程序是参照 IEEE 单精度浮点标准定义了一个 32 位自定义浮点类。

该类的定义如下：

```
//定义浮点类
class FloatType : public Str2Binary
{
public:
    FloatType();
    FloatType(string String);                                    //当以字符串格式输入数值的时候，调用该构造函数
    ~FloatType();
    void FloatInput(string str);                                 //输入函数，输入数值格式为字符串
    void printFloatObj(FloatType Data);                          //将自定义的浮点类以二进制以及十进制的形式打印
    friend FloatType operator + (const FloatType& a, const FloatType& b);
    //重载操作符‘+’
    friend FloatType complement(FloatType OBJ);
    //取补码运算
    friend FloatType operator -(const FloatType &a, const FloatType& b);     //重载操作符‘-’

private:
    unsigned int sign;                                           //use for sign bit, just can be 0 or 1;
    int exp[8];                                                  //use for exponent,
    unsigned int mant[23];                                       //use for mantissa,
//用于自定义浮点类运算的辅助函数
    void mantMovRight(int bits);                                 //尾数右移函数，用于对阶
    void mantMovLeft(int bits);                                  //尾数左移函数，用于将运算结果标准化
    void copyexp( FloatType &a);                                 //指数部分复制函数
    void expAdd(int num);                                        //指数部分加上num
    int mantAdd(FloatType &a, FloatType &b,int Flag);            //尾数部分相加函数
    void mantAdd(int num);                                       //尾数部分加上num
    void Binary2FloatType();                                     //用于将数值的二进制代码转化为标准的浮点二进制代码
    float FloatType2float(FloatType Data);                       //用于将FloatType的二进制代码转化为float数据，该函数是为了方面便验算
};
```

如上所示，该类还继承了一个叫 Str2Binary 的类。定义 Str2Binary 的类主要

是因为题目要求不能定义 float 类型的变量。因此，数据的输入只能以字符串的形式。该类的主要作用就是定义将字符串转化为二进制代码的方法。该类的定义如下：

```
class Str2Binary
{
public:
    Str2Binary();
    Str2Binary(string str);
    ~Str2Binary();
    void input(string str);
protected:
    int *Int_Binary = NULL;
    int *Dec_Binary = NULL;
    int sign_ = 0;
    int DecBinaryLen;
    int IntBinaryLen;
private:
    string inputdata;
    int calculateSpace(int n);
    void Str2BinaryTran();
    int DecPartMulti_2(int *DecPart,int DecLen);
    int IntPartDivi_2(int *IntPart, int IntLen);
    int* LenCountAndCopy(string &Int_str,string& Dec_str);
};
```

由于字符串转化为二进制代码并不是本题的要求，只是一个辅助手段，故在此不做过多介绍。

下面主要介绍该程序是如何使用浮点二进制代码实现浮点数的加减的。

对于浮点加法，该程序重载了运算‘+’操作符。对于浮点二进制代码的加法，跟一般二进制代码的加法有比较大的差别。对于两个加数，首先要判断两数的阶次是否相等，如果相等，当然可以直接尾数相加；但是，如果不等的话，就必须要先将两个数的阶数调成一样的，这个过程叫做对阶（该程序对阶的方法是将阶数较低的尾数右移 n 位，然后指数加上 n），然后再进行尾数相加。当然两数相加，还有可能会有进位之类的（并且由于 IEEE 标准下的浮点格式的尾数的整数部分是为以 1 的，所以如果两个都是正数相加，必然会向高位进位，但是又没有更高的位了，所以只能尾数右移，指数加 1）。对于两个不同符号的数相加的处理更是会出现比较极端的情况，如两个绝对值相等的很大的数符号是相反的，这时候，他们相加之后就等于零了，对于这种情况，就需要对尾数进行左移操作，指数减去相应的值。当然，尾数相加采用补码的方式。具体代码如下：

```
//重载操作符‘+’
FloatType operator+(const FloatType & a, const FloatType & b)
{
    FloatType temp_a, temp_b, temp_c;
    temp_a = a;
    temp_b = b;
    int exp_a=0, exp_b=0;
    int i = 0;
    int Flag_mantMov = 0;
    for ( i = 0; i < 8; i++)
    {
        exp_a = exp_a * 2 + temp_a.exp[i];
        exp_b = exp_b * 2 + temp_b.exp[i];                    //exp[0] is MSB
    }
    int isCarry = 0;
    if (exp_a==exp_b)
    {
        temp_a = complement(temp_a);                                    //取补
码
        temp_b = complement(temp_b);
        isCarry = temp_c.mantAdd(temp_a, temp_b, Flag_mantMov);
    //mantissa process
        temp_c.copyexp(temp_a);                                     //exp
process
    }
    else                                                          //如果不是同阶次
的，要先右移成同阶次再计算
    {
        int temp;
        temp = exp_a - exp_b;
        Flag_mantMov = temp;
        if (temp > 0&&temp<23)
        {
            temp_b.mantMovRight(temp);
            temp_b.mant[temp-1] = 1;
            temp_a = complement(temp_a);                                //取补
码
            temp_b = complement(temp_b);
            isCarry = temp_c.mantAdd(temp_a, temp_b, Flag_mantMov);
    //mantissa process
            temp_c.copyexp(temp_a);
        }
```

```
        else if(temp>-23&&temp<0)
        {
            temp = -temp;
            temp_a.mantMovRight(temp);
            temp_a.mant[temp - 1] = 1;
            temp_a = complement(temp_a);                                    //取补码
            temp_b = complement(temp_b);
            isCarry = temp_c.mantAdd(temp_a, temp_b, Flag_mantMov);
    //mantissa process
            temp_c.copyexp(temp_b);
        }
        else
        {
            if (temp < -23)
                temp_c = complement(temp_b);
            if (temp > 23)
                temp_c =complement(temp_a);
        }
    }
    if(isCarry==-127)       //说明已经是零了
    {
        for (int i = 0; i < 8; i++)
            temp_c.exp[i] = 0;
    }
    else
        temp_c.expAdd(isCarry);
    temp_c = complement(temp_c);                                    //再取一次补码，即得
到原码
    return FloatType(temp_c);
}
```

该函数的 isCarry 由 mantAdd()（尾数相加）函数返回的。mantAdd()函数具体如下：（由于该函数处理的尾数相加情况比较多，所以函数体比较长，但是思路是比较清晰的，在这里只把部分放出来）

```
int FloatType::mantAdd(FloatType & a, FloatType & b,int Flag)
{
//将23位mant按位相加，为进一步模拟机器的运算，整理采用条件判断的方法进行，而没有采用
‘/’或‘%’操作符
    int isCarry=0;
    for (int i = 0; i < 23; i++)
    {
        if (isCarry == 0)
        {
            if (a.mant[22 - i] + b.mant[22 - i] > 1)
```

```
            {
                isCarry = 1;
                mant[22 - i] = 0;
            }
            else mant[22 - i] = a.mant[22 - i] + b.mant[22 - i];
        }
        else
        {
            if (a.mant[22 - i] + b.mant[22 - i] > 1)
            {
                mant[22 - i] = 1;
                isCarry = 1;
            }
            else if(a.mant[22 - i] + b.mant[22 - i]==1)
            {
                mant[22 - i] = 0;
                isCarry = 1;
            }
            else
            {
                isCarry = 0;
                mant[22 - i] = 1;
            }
        }
    }
```

以上就是将尾数相加的部分（还没有处理最高位的进位问题）

由于最高位的进位问题情况比较多，这里只贴出一种情况的代码；

```
    else if (a.sign == 0 || b.sign == 0)     //有一个是负数
    {
        if (isCarry == 1)                    //尾数补码最高位有进位
        {
            if (Flag == 0)                   //两者同阶次
            {
                sign = 0;
                int i = 0;
                for (i = 0; i < 23&&mant[0]==0; i++)
                    mantMovLeft(1);
                mantMovLeft(1);
                isCarry = -(i+1);
                if (i + 1 >= 23)
                    isCarry = -127;      //左移超过 23 位，说明结果应该为零，所以应该把指数值为 0（返回-127 作为标志）
```

以上就是对其中一种极端情况的处理。

有了操作‘+’，‘-’的实现就很简单了，因为可以调用加法来处理。

```
//重载操作符‘-’
FloatType operator-(const FloatType & a, const FloatType & b)
{
    FloatType temp_a, temp_b, temp_c;
    temp_a = a;
    temp_b = b;
    if (temp_b.sign == 1)
        temp_b.sign = 0;
    else
        temp_b.sign = 1;
    temp_c = temp_a + temp_b;
    return FloatType(temp_c);
 }
```

这个程序的关键内容就是如此了。下面对该程序进行一些测试：

加法测试：a，b 为加数，c 为结果

a	0.0	0 00000000 0000 0000 0000 0000 0000 000
b	0.0	0 00000000 0000 0000 0000 0000 0000 000
c	0.0	0 00000000 0000 0000 0000 0000 0000 000
a	123456.123	0 10001111 1110 0010 0100 0000 0010 000
b	-123456.123	1 10001111 1110 0010 0100 0000 0010 000
c	0.0	0 00000000 0000 0000 0000 0000 0000 000
a	99999999999.99	0 10100011 0111 0100 1000 0111 0110 111
b	12456789211.25	0 10100000 0111 0011 0011 1101 1100 001
c	112456785920.0	0 10100011 1010 0010 1110 1111 0010 011
a	-4598745.123	1 10010101 0001 1000 1010 1111 0110 010
b	-9825613.456	1 10010110 0010 1011 1101 1010 1001 101
c	-14424358.00	1 10010110 1011 1000 0011 0010 0100 110
a	0.0000125	0 01101110 1010 0011 0000 0000 0000 000
b	0.0000895	0 01110001 0111 0111 0110 0000 0000 000
c	0.000102	0 01110001 1010 1011 1100 0000 0000 000

减法测试：a 为被减数，b 为减数，c 为结果

a	0.0	0 00000000 0000 0000 0000 0000 0000 000
b	0.0	0 00000000 0000 0000 0000 0000 0000 000
c	0.0	0 00000000 0000 0000 0000 0000 0000 000
a	123456.123	0 10001111 1110 0010 0100 0000 0010 000
b	-123456.123	1 10001111 1110 0010 0100 0000 0010 000
c	246912.25	0 10010000 1110 0010 0100 0000 0010 000
a	99999999999.99	0 10100011 0111 0100 1000 0111 0110 111
b	12456789211.25	0 10100000 0111 0011 0011 1101 1100 001
c	87543209984.00	0 10100011 0100 0110 0001 1111 1011 011
a	-4598745.123	1 10010101 0001 1000 1010 1111 0110 010
b	-9825613.456	1 10010110 0010 1011 1101 1010 1001 101
c	3129716.000	0 10010100 0111 1110 0000 1011 1010 000
a	0.0000125	0 01101110 1010 0011 0000 0000 0000 000
b	0.0000895	0 01110001 0111 0111 0110 0000 0000 000
c	-0.000077	1 01110001 0100 0011 0000 0000 0000 000