浙江大学实验报告

实验名称：浮点数的表示与算术运算算法

学生姓名：翁罗轩 专业：软件工程 学号：3170103226

实验日期：2019 年 5 月 8 日

一、 实验目的

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。具体需要实现以下六个函数：

```
//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);
```

二、 实验原理

2.1 浮点数标准格式转换为字符串

首先通过移位操作分别得到该浮点数的尾数 m 与阶码 e，然后通过 itoa 函数将尾数以二进制形式转换为一个字符串，注意，若不足 23 位，需要在左端补 0。根据阶码的值确定该浮点数是小于 1 还是大于 1，然后将其还原回二进制形式，并保留小数点的位置，保存在字符数组 binary_str 中。至此，就可以通过该数组还原该浮点数的整数部分和小数部分了。小数点左边的数依次乘以 2 的正数次幂，小数点右边的数依次乘以 2 的负数次幂，得到 d_int 和 d_float，相加以后在最前面加上符号位，就得到了该浮点数。最后通过 gcvt 函数将这个浮点数转换为字符串，保存在 result_str 中，输出即可。

2.2 字符串转换为浮点数标准格式

我在这里采用的方法比较麻烦，但相对还算稳定，在绝大部分情况下能得到

正确结果。具体思路是：分别将传入字符串中小数点左边（整数部分）与右边（小数部分）提取出来（主要是针对字符的一些操作），保存在整型常量 str_int 与浮点数常量 str_float 中。现在得到了形如 xxx.xxx 格式的“数”，考虑到后续需要从中提取阶码与尾数，为方便起见，将其以二进制形式转换为一个字符串。使用 itoa 函数转换整数部分，加上小数点后，每次乘 2 取整转换小数部分，并转换为字符存入字符数组 binary_str 中。如果是无限循环小数，则只取其前 45 位。

获得阶码 e 的方法是：若整数部分为 0，则 e = -（小数点后面 0 的个数 + 1） + 127；若整数部分非 0，则 e = （整数部分位数 - 1） + 127。将 e 转换为字符串，若不足 8 位，左端补 0。

获得尾数 m 的方法是：若整数部分非 0，直接取小数点后面 23 位即可；若整数部分为 0，则首先移动小数点，找到其正确的位置后再取后面 23 位。注意，若不足 23 位，右端补 0。m 同样保存在字符串中。

至此，阶码 e 与尾数 m 的字符串都得到了，将它们拼接起来，并在首位加上符号位 s 即可得到结果字符串 result_str，将其转换成十六进制无符号整数即为结果。

2.3 加减

加法和减法的原理类似，因此这里只介绍加法的原理。

首先通过移位操作，分别得到两个浮点数的阶码与尾数。采用低阶向高阶对齐的方法，将低阶的尾数进行向右移位操作。此时，两个尾数已经对齐了，直接对它们进行加减操作即可（同号为加，异号为减）。

需要注意的是，异号相加可能导致尾数的首位不再是 1，此时只要将其向左移位，同时减小阶码的值即可化为标准格式。同号相加可能导致进位，此时则通过向右移位，同时增大阶码的值即可化为标准格式。

最后，将符号位、阶码、尾数拼接起来即得到结果。

对于减法，唯一不同的地方在于符号位的判断，需要根据符号与绝对值大小关系分情况讨论。

2.4 乘除

除法和乘法的原理完全一样，因为除法可以化为被除数乘以除数的倒数（通过 union 实现），因此这里只介绍乘法的原理。

与加减法一样，首先通过移位操作得到阶码和尾数。将两个阶码相加，并减去 127 即可得到积的阶码（后面可能会变）。然后将尾数相乘，具体为乘数每次右移一位，根据其最低位的情况来判断是否要在积里加上被乘数（0 则不加，1 则加）。相应地，每次运算前，积也需要右移一位。

同样地，乘法也可能导致进位，采用向右移位、增大阶码的方法即可化为标准格式。

最后，将符号位（同号为 0，异号为 1）、新的阶码和新的尾数拼接起来即得到结果。

2.5 特殊情况的处理

针对±INF、0、NaN 等特殊情况，本程序做了相应的特殊处理。

在 ftoa 函数中，当输入为 0x7F800000 或 0xFF800000 或 0 时，给出相应的

输出，如下图所示：

```
//special cases
if (d == 0x7F800000)
{
    strcpy(result_str, "INF");
    return result_str;
}
else if (d == 0xFF800000)
{
    strcpy(result_str, "-INF");
    return result_str;
}
else if (d == 0)
{
    strcpy(result_str, "0");
    return result_str;
}
```

在 atof 函数中，当输入为某些特殊情况时，给出相应的特殊输出，如下图所示：

```
//special cases
if (string[0] == '0' && string[1] == '\0')
    return 0;
if (strcmp(string, "INF"))
    return 0x7F800000;
if (strcmp(string, "-INF"))
    return 0xFF800000;
```

```
//special cases
if (s == 0 && e == 255 && atol(m_str) == 0)
{
    printf("INF\n");
    return 0;
}
else if (s == 1 && e == 255 && atol(m_str) == 0)
{
    printf("-INF\n");
    return 0;
}
else if (e == 255 && atol(m_str) != 0)
{
    printf("NaN\n");
    return 0;
}
```

为简便起见，在加、减、乘、除等运算函数中，暂未考虑这些特殊情况的处理。

2.6 适用范围

虽然浮点数的范围是 10 的正负 38 次方，但本程序所能表示的范围比较有限。经测试，范围约为 10 的正负 9 次方，一旦需要表示的数超过这个范围，就会出现不同程度的精度损失情况。

关于原因，我认为不是程序本身的问题，而是单精度浮点数本身的有效位数所限制的。比如，输入一个多位浮点数 0.0000000005 时，我对程序进行了调试：

```
118     int times = 10;
119     while (string[i] != '\0')
120     {
121         str_float += (float)(string[i] - '0') / times;
122         times *= 10;  已用时间 <= 1ms
123         i++;
124     }
125
126     char binary_str[100];
127
128     itoa(str_int, binary_str, 2);
129
130     if (str_int != 0)
131     {
132         e = strlen(binary_str) - 1 + 127;
133     }
134
135     int ptr = strlen(binary_str);
136     binary_str[ptr++] = '.';
```

100 %

自动窗口

名称	值	类型
i	11	int
str_float	3.54593488e-09	float
string	0x000001a710294c00 "0.0000000005"	char *
string[i]	53 '5'	char
times	1410065408	int

可见，当程序执行到读取字符串最后一个字符时（即‘5’），虽然读取成功了（即 string[i] == ‘5’），但 str_float 却不再是相应的数值。所以我认为应该是超出精度范围了。

对于大数，如 123456789，传入 ftoa 函数的标准格式是正确的，但由于要从中提取出尾数和阶码，只能通过移位操作，这样做必然会导致精度损失，因此转换出来的字符串与原浮点数有不同程度上的出入。

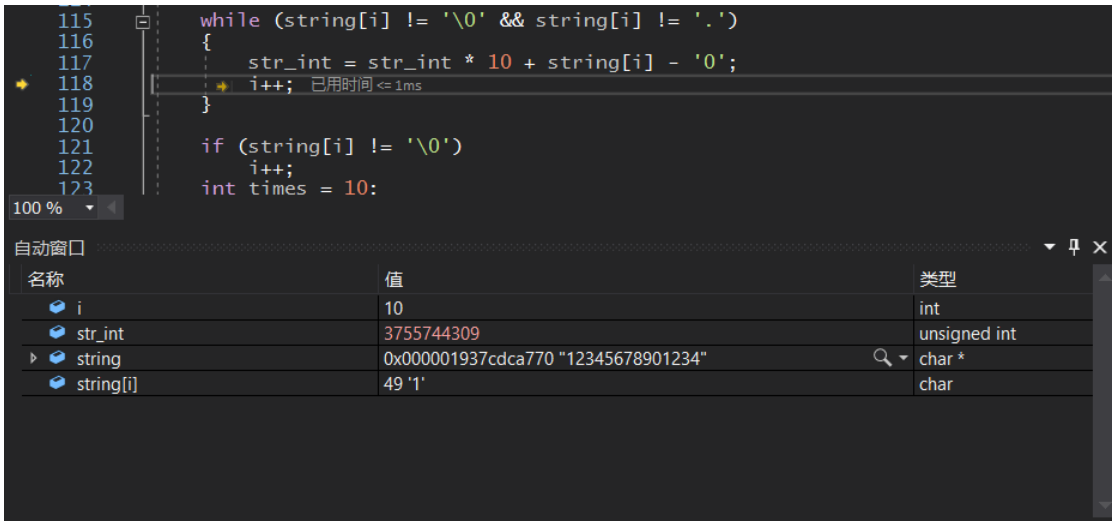
对于更大的数，如 12345678901234，我进行了调试：

```
115     while (string[i] != '\0' && string[i] != '.')
116     {
117         str_int = str_int * 10 + string[i] - '0';
118         i++;
119     }  已用时间 <= 1ms
120
121     if (string[i] != '\0')
122         i++;
123     int times = 10;
```

100 %

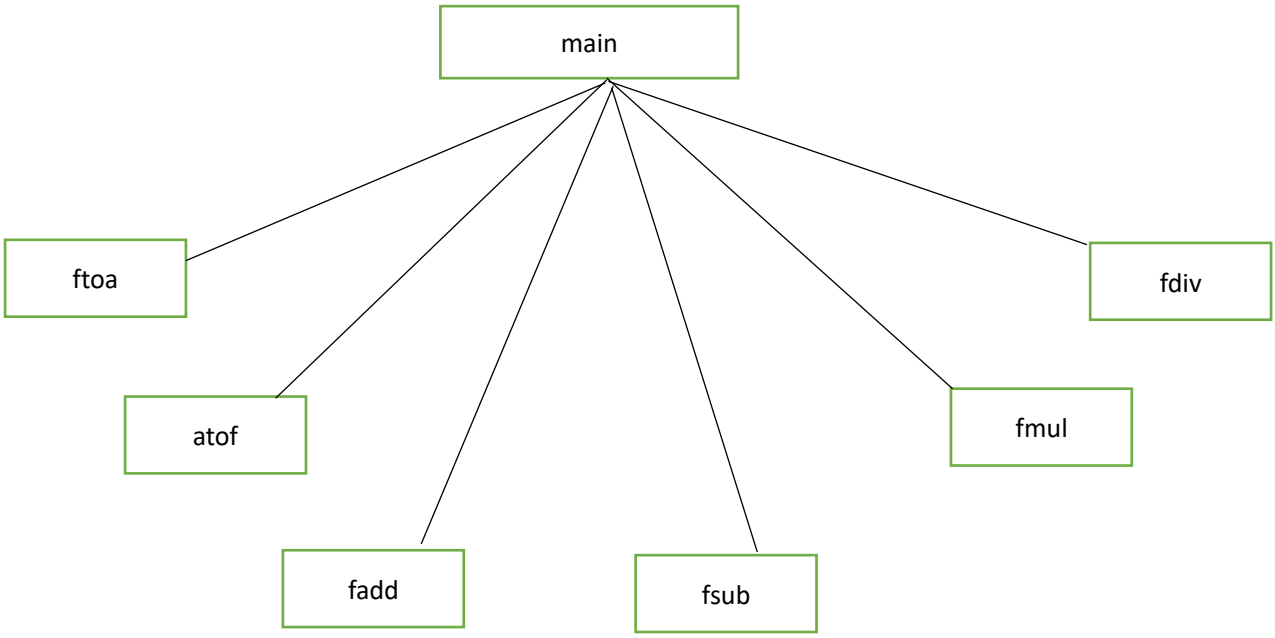
自动窗口

名称	值	类型
i	10	int
str_int	1234567890	unsigned int
string	0x000001937cdca770 "12345678901234"	char *
string[i]	49 '1'	char



可见，读到’1’时，str_int 不是预期的结果 12345678901，而是 3755744309，这显然是发生了溢出。

2.7 程序框图



三、 使用方法

1. 本代码在 Visual Studio 2017 中能正常编译运行。

2. 运行程序，根据提示选择对应的操作。需注意：将浮点数格式化时，只能输入 xxx.xxx 的格式，而不支持科学计数法，但支持±INT 与 0 的特殊情况；将 IEEE754 格式重新转换成浮点数字符串时，只能输入十六进制。

3. 加减乘除操作同上，不支持科学计数法输入。

4. 当要输出的数小于 10 的负 2 次方时，程序会自动给出其科学计数法表示（这是由所调用的 gcvt 函数本身的特性决定的）。

5. 某些特殊情况，如 0、±INF、NaN 等，不支持四则运算，但支持将它们的 IEEE754 格式转换回来。比如，输入 0x7F800000，会输出 INF。

四、 测试结果

4.1 浮点数标准格式转换为字符串

```
Choose the operation you want to take:
1.Change a string into float number in IEEE754 format.
2.Change a float number in IEEE754 format into string.
3.Add.
4.Subtract.
5.Multiply.
6.Divide.
0.Exit.
1
Enter a string.
4.55567
0X4091C80C
```

4.2 字符串转换为浮点数标准格式

```
Choose the operation you want to take:
1.Change a string into float number in IEEE754 format(hex).
2.Change a float number in IEEE754 format into string.
3.Add.
4.Subtract.
5.Multiply.
6.Divide.
0.Exit.
2
Enter a float number in IEEE754 format(hex).
0X4091C80C
4.555669785
```

4.3 加

```
Choose the operation you want to take:
1.Change a string into float number in IEEE754 format(hex).
2.Change a float number in IEEE754 format into string.
3.Add.
4.Subtract.
5.Multiply.
6.Divide.
0.Exit.
3
Enter the first float number.
1.5
Enter the second float number.
-2.3
1.5(0X3FC00000) + -2.3(0XC0133333) = -0.7999999523(0XBF4CCCCC)
```

4.4 减

```
Choose the operation you want to take:
1.Change a string into float number in IEEE754 format(hex).
2.Change a float number in IEEE754 format into string.
3.Add.
4.Subtract.
5.Multiply.
6.Divide.
0.Exit.
4
Enter the first float number.
5.88
Enter the second float number.
6.7
5.88(0X40BC28F5) - 6.7(0X40D66666) = -0.8200001717(0XBF51EB88)
```

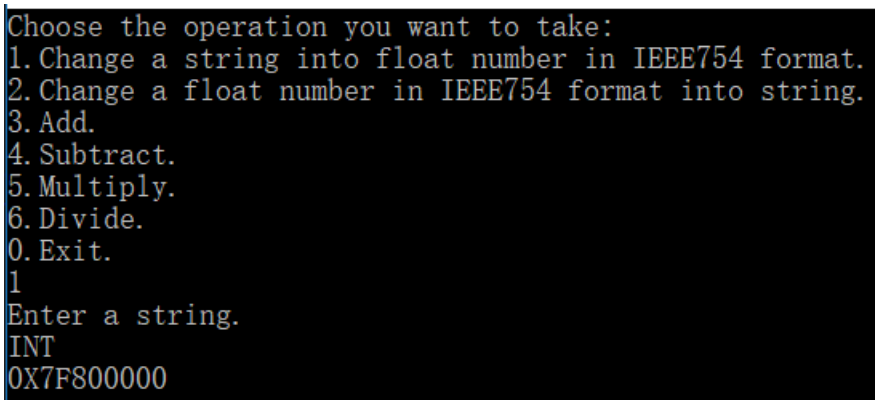
4.5 乘

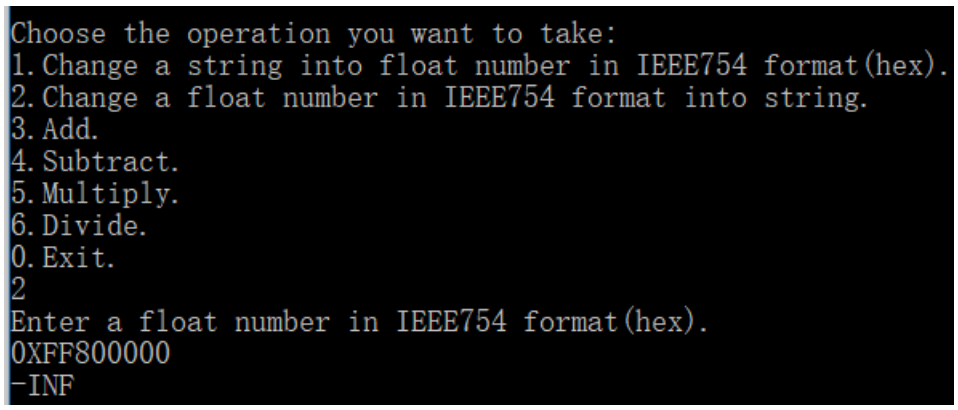
```
Choose the operation you want to take:
1.Change a string into float number in IEEE754 format(hex).
2.Change a float number in IEEE754 format into string.
3.Add.
4.Subtract.
5.Multiply.
6.Divide.
0.Exit.
5
Enter the first float number.
3.55
Enter the second float number.
2.7
3.55(0X40633333) * 2.7(0X402CCCCC) = 9.584999084(0X41195C28)
```

4.6 除

```
Choose the operation you want to take:
1.Change a string into float number in IEEE754 format(hex).
2.Change a float number in IEEE754 format into string.
3.Add.
4.Subtract.
5.Multiply.
6.Divide.
0.Exit.
6
Enter the first float number.
3.54
Enter the second float number.
0.6
3.54(0X40628F5C) / 0.6(0X3F19999A) = 5.899999619(0X40BCCCCC)
```

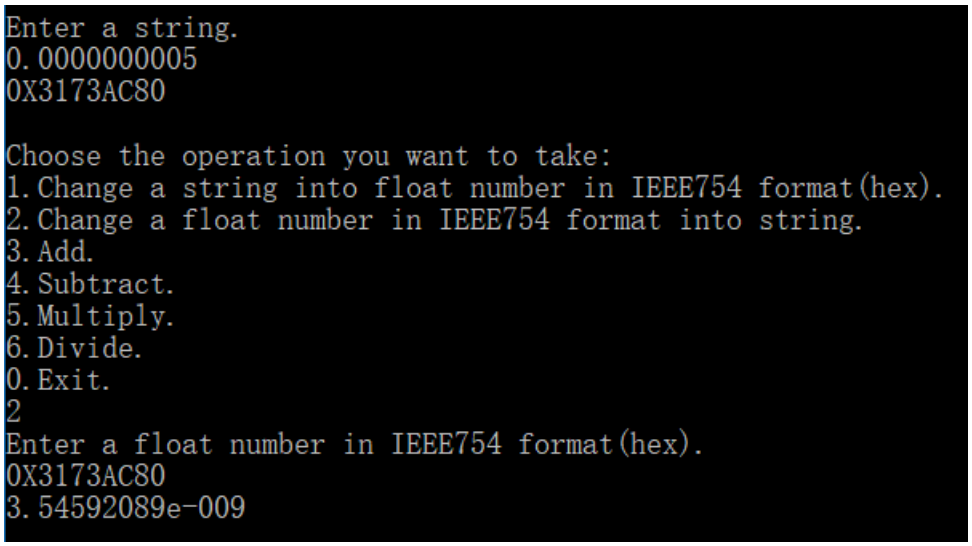
4.7 特殊情况输出

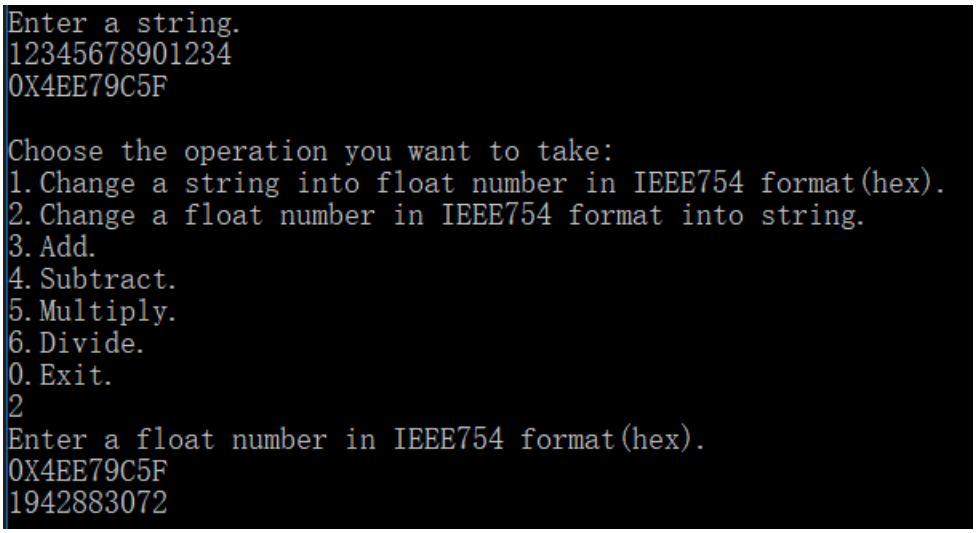




4.8 大数小数验证

在本程序中，当输入的数超过范围时（粗略估计约为 10^{-9}-10^{9}），会输出错误（或不精确）的结果，如下图所示：




Enter a string.
12345678901234
0X4EE79C5F
Choose the operation you want to take:
1.Change a string into float number in IEEE754 format(hex).
2.Change a float number in IEEE754 format into string.
3.Add.
4.Subtract.
5.Multiply.
6.Divide.
0.Exit.
2
Enter a float number in IEEE754 format(hex).
0X4EE79C5F
1942883072

原因分析见第二部分。