在C语言中，处理浮点数时，尽量不要让一个很大的数去加上一个很小的数，因为这样容易产生误差。

例：程序float\_test

float a = 12345.789e5;

float b = a + 20;

printf("%f\n", a); // 1234578944.000000

printf("%f\n", b); // 1234578944.000000

例：

在C中，除法会出现下面的现象：

double x = 4 / 512; // x = 0.000000

double x1 = 4 / 512.0; // x1 = 0.0078125

double x2 = 4.0 / 512; // x2 = 0.0078125

double x3 = 4.0 / 512.0; // x3 = 0.0078125

所以，如果希望得到准确的浮点型数据，被除数和除数之间必须至少有一个为浮点型数据。

printf("%f\n", (float)3 / 4); // 0.750000

C/C++中，浮点数的存储遵循IEEE 754标准。

IEEE 754规定：

32位的浮点数：最高位表示符号位，之后的8位表示指数位，剩下的23位表示有效数字



64位的浮点数：最高位表示符号位，之后的11位表示指数位，剩下的52位表示有效数字。



IEEE 754规定，任意一个浮点数V可以表示成下面的形式：



举例来说，十进制5.0，表示成二进制为101.0，这样，s = 0, m = 1.01，e = 2.（注意：这里的指数表示指的的二进制的指数，而并非十进制的指数）

IEEE 754中，关于m和e有一些特别规定：

在绝大多数情况下（后面将会提到例外的情况），，也就是说m可以表示成1.xxxx的形式，这样保存时可以将1舍去，只保存xxxx，这样做可以节省1位有效数字，例如32位浮点数时，m有23位，而实际可以保存24位有效数字。

指数e的规定：

首先e为一个无符号整数（其没有符号位），对于32位浮点数，其范围为0-255，对于64位浮点数，其范围为0-2047.但在科学计数法中，e可以为负值，所以IEEE 754规定，在存储浮点数时，e中保存的值为真实值+中间数，对于32位浮点数，中间数为127，对于64位浮点数，中间数为1023.

例如，真实值e为，在保存时，必须保存成10+127 = 137，即e在存储时为10001001.

此外，e还有以下3种规定：

1. e不全为0或e不全为1，此时称浮点数为规格化形式。根据存储的二进制计算浮点数时，首先将e中存储的值减去127（或1023），然后在m存储的值前面加上1.，从而构成1.xxxx；
2. e全为0，此时浮点数为非规格化形式。此时指数e的值为1-127（或1-1023），有效数字m之前不在补1，而补0，构成0.xxxx，这样做是为了表示以及很接近0的数字。
3. e全为1，如果m全为0，则表示（正负无穷大），如果m不全为0，则表示这不是一个数（NaN）。

例：程序float\_storage\_test1

int number = 9;

float\* f\_ptr = &number;

printf("number = %d\n", number);

printf("\*f\_ptr = %f\n", \*f\_ptr);

输出结果为：

number = 9

\*f\_ptr = 0.000000

从结果可以看出，浮点数输出时9被输出成了0.000000，原因如下：

整型变量number在内存中占有4个字节，存储成二进制为：0x00000009，根据IEEE 754规定的浮点数的存储规则：

s = 0，e全部为0，m = 000 0000 0000 0000 0000 1001，合成浮点数为：



得到的浮点数v是一个非常接近于0的正数，所以显示为0.000000

接着上面的例子：

\*f\_ptr = 9.0;

printf("number = %d\n", number);

printf("\*f\_ptr = %f\n", \*f\_ptr);

输出为：

number = 1091567616

\*f\_ptr = 9.000000

可以看出，将浮点数9.0赋给\*f\_ptr后，输出的浮点数为9.0，而按整数格式输出时，则变成了1091567616，原因如下：

浮点数9.0占有4个字节，转换成二进制时为1001.0，即：

具体在内存中存储如下：

1. 符号位为0；
2. 指数部分存储的为：3+127 = 130，即1000 0010
3. 有效数字部分为：舍去1，将001存储在前3位，后面补20个0，即：001 0000 0000 0000 0000 0000

合在一起为：

0 10000010 00100000000000000000000

将上述的二进制转换为十进制就为：1091567616

接着上面的例子：

\*f\_ptr = 12.34;

printf("number = %d\n", number);

printf("\*f\_ptr = %f\n", \*f\_ptr);

输出结果为：

number = 1095069860

\*f\_ptr = 12.340000

浮点数12.34转换成二进制为： 小数部分转换成二进制有无穷多位，产生了进位。

1100.01010111000010100100

写成指数形式为：



符号为0

指数部分为：3 + 127 = 130 即1000 0010

有效数字部分为：

合在一起为：

0 10000010 10001010111000010100100

转换成十进制为：1095069860

如果按照下面的方式编写程序，则结果是未知的：

float fval = 9.1;

printf("fval = %d\n", fval);

这种情况下，不会按照上面讲述的IEEE 754的标准将浮点数转换为二进制，再按整数输出，而是会出现数据丢失，结果未知。

将浮点数的二进制转换为浮点数：

二进制转换为十进制时，小数点后的二进制转换为对应的二进制的值，其中n为位数

十进制5.0，表示成二进制为101.0，这样，s = 0, m = 1.01，e = 2，即：



将上述表达式重新转换成十进制：

十进制12.34，表示成二进制为：



重新计算成浮点数为：

浮点型float无法取代int型的原因：

例：

int number = 9;

float\* f\_ptr = &number;

int a = 1234567890;

\*f\_ptr = 1234567890;

printf("number = %d\n", number);

printf("a = %d \n", a);

printf("\*f\_ptr = %f\n", \*f\_ptr);

输出：

a = 1234567890

\*f\_ptr = 1234567936.000000

可以看出：int型的定点数可以精确的表示出1234567890，而浮点型确无法精确的表示1234567890.

原因如下：浮点数1234567890的二进制表示为：

100 1001 1001 0110 0000 0010 1101 0010

即：(有效数字部分发生了进位，最多24位) 

再转换为十进制为：



可以看出，前7位是正确的，后3位是无效的，这跟float型的精度有关。具体参考：

<https://blog.csdn.net/qq_16137569/article/details/79508091>

在C/C++中，float型变量只能精确到小数点后6位。

例：程序output\_test1

#include <iostream>

using std::cout;

using std::endl;

int main(int argc, char\* argv[])

{

float fval1 = 9.87654311;

float fval2 = 9.87654322;

// 结果为fval1 = fval2

// 可见float型只能精确到小数点后6位

if (fval1 < fval2)

{

cout << "fva1 < fval2" << endl;

} else if (fval1 == fval2)

{

cout << "fval1 = fval2" << endl;

} else

{

cout << "fval1 > fval2" << endl;

}

// double型可以精确到小数点后15位 dval1=dval2

double dval1 = 9.8765432187654321;

double dval2 = 9.8765432187654322;

if (dval1 < dval2)

{

cout << "dva1 < dval2" << endl;

} else if (dval1 == dval2)

{

cout << "dval1 = dval2" << endl;

} else

{

cout << "dval1 > dval2" << endl;

}

return 0;

}