

1. 整数在内存中的存储

整数的2进制表示方法有三种，即**原码、反码和补码**

三种表示方法均有符号位和数值位两部分，符号位都是用0表示“正”，用1表示“负”，而数值位最高位的一位是被当做符号位，剩余的都是数值位。

正整数的原、反、补码都相同。负整数的三种表示方法各不相同。

原码：直接将数值按照正负数的形式翻译成二进制得到的就是原码。

反码：将原码的符号位不变，其他位依次按位取反就可以得到反码。

补码：反码+1就得到补码。

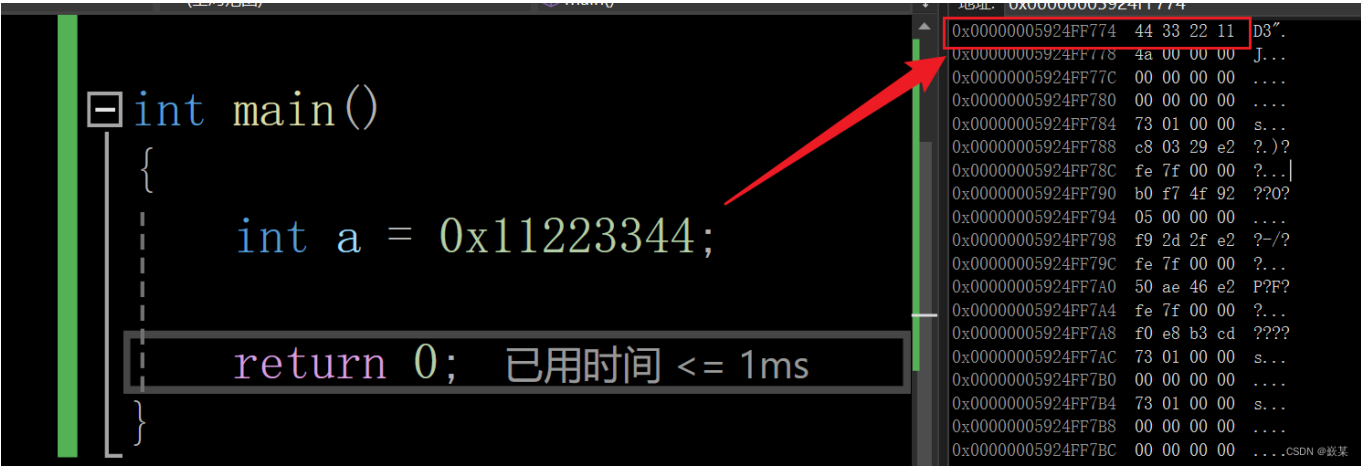
对于整形来说：数据存放**内存中其实存放的是补码**。

为什么呢？

在计算机系统中，数值一律用补码来表示和存储。原因在于，使用补码，可以将符号位和数值域统一处理；同时，加法和减法也可以统一处理（**CPU只有加法器**）此外，补码与原码相互转换，其运算过程是相同的，不需要额外的硬件电路。

2. 大小端字节序和字节序判断

我们先来看一个细节：



调试的时候，我们可以看到在a中的 0x11223344 这个数字是按照字节为单位，倒着存储的。这是为什么呢？

2.1 什么是大小端？

其实超过一个字节的数在内存中存储的时候，就有存储顺序的问题，按照不同的存储顺序，我们分为大端字节序存储和小端字节序存储，下面是具体的概念：

- 大端（存储）模式：是指数据的低位字节内容保存在内存的高地址处，而数据的高位字节内容，保存在内存的低地址处。
- 小端（存储）模式：是指数据的低位字节内容保存在内存的低地址处，而数据的高位字节内容，保存在内存的高地址处。

2.2 为什么有大小端？

这是因为在计算机系统中，我们是以字节为单位的，每个地址单元都对应着一个字节，一个字节为8bit位，但是在C语言中除了8bit的 char 之外，还有16bit的 short 型，32bit的 long 型（要看具体的编译器），另外，对于位数大于8位的处理器，例如16位或者32位的处理器，由于寄存器宽度大于一个字节，那么必然存在着一个如何将多个字节安排的问题。因此就导致了大端存储模式和小端存储模式。

例如：一个 16bit 的 short 型 x，在内存中的地址为 0x0010，x 的值为 0x1122，那么0x11 为高字节，0x22 为低字节。对于大端模式，就将 0x11 放在低地址中，即 0x0010 中，0x22 放在高地址中，即 0x0011 中。小端模式，刚好相反。我们常用的 X86 结构是小端模式，而**KEIL C51** 则为大端模式。很多的ARM，DSP都为小端模式。有些ARM处理器还可以由硬件来选择是大端模式还是小端模式。

3. 浮点数在内存中的存储

在了解浮点数在内存中的存储之前先看一下这段代码的输出：

```
#include
int main()
{
    int n = 9;
    float *pFloat = (float *)&n;

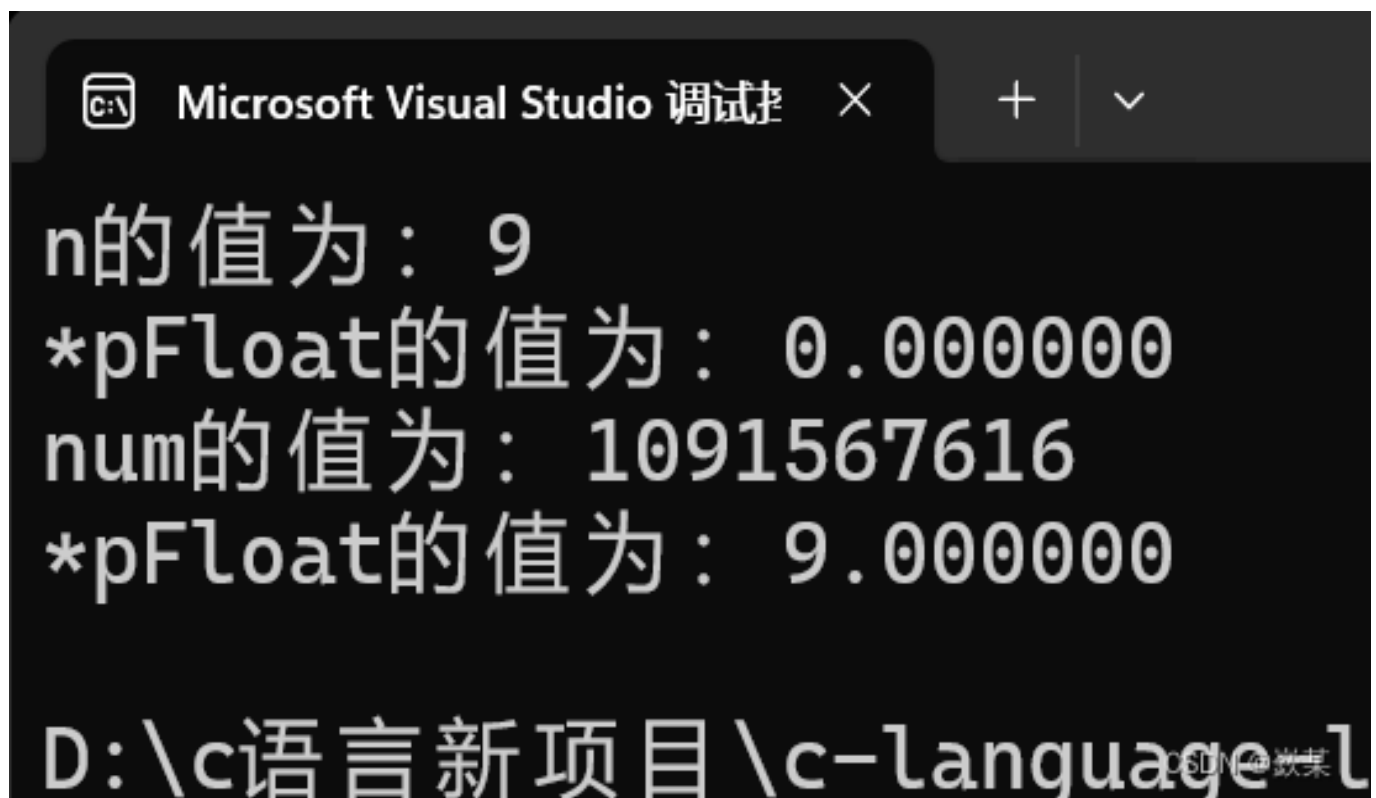
    printf("n的值为: %d\n",n);
    printf("*pFloat的值为: %f\n",*pFloat);

    *pFloat = 9.0;

    printf("num的值为: %d\n",n);
    printf("*pFloat的值为: %f\n",*pFloat);

    return 0;
}
```

结果：

A screenshot of the Microsoft Visual Studio debug console. The window title is "Microsoft Visual Studio 调试". The output text is as follows:
n的值为: 9
*pFloat的值为: 0.000000
num的值为: 1091567616
*pFloat的值为: 9.000000
At the bottom, the file path "D:\c语言新项目\c-language-l" is visible, followed by a small logo and the text "OSDN 某某".

```
C:\V Microsoft Visual Studio 调试 × + ∨
n的值为: 9
*pFloat的值为: 0.000000
num的值为: 1091567616
*pFloat的值为: 9.000000
D:\c语言新项目\c-language-l OSDN 某某
```

直接看肯定是看不懂的，我们先了解浮点数是如何在内存存储的。

根据国际标准IEEE（电气和电子工程协会）754，任意一个二进制浮点数V可以表示成下面的形式：

$$V = (-1)^S * M * 2^E$$

CSDN @某某

- $(-1)^S$ 表示符号位，当S=0，V为正数；当S=1，V为负数
- M表示有效数字，M是大于等于1，小于2的
- 2^E 表示指数位

举例来说：

十进制的5.0，写成二进制是 101.0，相当于 1.01×2^2 。那么，按照上面V的格式，可以得出S=0，M=1.01，E=2。

十进制的-5.0，写成二进制是 -101.0，相当于 -1.01×2^2 。那么，S=1，M=1.01，E=2

IEEE754规定：

对于32位的浮点数，最高的1位存储符号位S，接着的8位存储指数E，剩下的23位存储有效数字M

对于64位的浮点数，最高的1位存储符号位S，接着的11位存储指数E，剩下的52位存储有效数字M

3.1 具体存储过程：

IEEE754对有效数字M和指数E，还有一些特别规定。

前面说过， $1 \leq M < 2$ ，也就是说，m可以写成 1.xxxxxx 的形式，其中 xxxxxx 表示小数部分。IEEE754规定，在计算机内部保存m时，默认这个数的第一位总是1，因此可以被舍去，只保存后面xxxxxx部分。比如保存1.01的时候，只保存01，等到读取的时候，再把第一位的1加上去。这样做的目的，是节省1位有效数字。以32位浮点数为例，留给M只有23位，**将第一位的1舍去**以后，等于可以保存24位有效数字。

至于指数E，情况就比较复杂

首先，E为一个无符号整数（unsigned int）

这意味着，如果E为8位，它的取值范围为0~255；如果E为11位，它的取值范围为0~2047。但是，我们知道，科学计数法中的E是可以出现负数的，所以IEEE754规定，存入内存时E的真实值必须再加上一个中间数，**对于8位的E，这个中间数是127；对于11位的E，这个中间数是1023**。比如， 2^{10} 的E是10，所以保存成32位浮点数时，必须保存成 $10 + 127 = 137$ ，即10001001。

3.2 浮点数读取过程

指数E从内存中取出还可以再分成三种情况：

E不全为0或不全为1

这时，浮点数就采用下面的规则表示，即指数E的计算值减去127（或1023），得到真实值，再将有效数字M前加上第一位的1。

比如：0.5的二进制形式为0.1，由于规定正数部分必须为1，即将小数点右移1位，则为 1.0×2^{-1} ，其阶码为 $-1 + 127$ (中间值) = 126，表示为01111110，而尾数1.0去掉整数部分为0，补齐0到23位
00000000000000000000000

则其二进制表示形式为：

1 0 01111110 00000000000000000000000

E全为0

这时，浮点数的指数E等于1-127（或者1-1023）即为真实值，有效数字M不再加上第一位的1，而是还原为0.xxxxxx的小数。这样做是为了表示 ± 0 ，以及接近于0的很小的数字。

0 00000000 00100000000000000000000

E全为1

这时，如果有效数字M全为0，表示 \pm 无穷大（正负取决于符号位s）

0 11111111 00010000000000000000000

本期博客到这里就结束了，如果有什么错误，欢迎指出，如果对你有帮助，请点个赞，谢谢！