系统整数分析

王子腾 3180102173 2020/2/29

问题一：

运行以下C语言程序：

```
#include <iostream>
#include <stdio.h>
int main(int argc, char** argv){
    int x=-2147483648;
    printf("%d, %d", x, -x);
}
```

分析以上程序的运行结果，为什么。

解答：

1. 实验运行结果为：

   ```
   -2147483648, -2147483648
   ```

2. 在计算机中，一个int整型变量占用4字节（即32位），根据整数的二进制存储格式，其能够存储的范围为-2^{31} ~ 2^{31}-1（即-2147483648~2147483647）。

   存储时，-2147483648的反码为1000 0000 0000 0000，取反时，进行的操作是先按位取反再加一，即：

   ```
     0111 1111 1111 1111
   + 0000 0000 0000 0001
   ---------------------
   = 1000 0000 0000 0000
   ```

   在这个过程中发生了进位，符号位由0变为1，而代表实际数字的位则被清零，此时发生了溢出

3. 由于计算机完成加法运算非常快，因此为加快底层运行速度，我们使用取反相加来代替减法，即反码，来代表取反，而此时则会出现全1位与全0位都代表零的情况，即：

   ```
   1111 1111 1111 1111 代表 -0
   0000 0000 0000 0000 代表 +0
   ```

   这会造成跨0运算产生误差，因此制订了补码原则，即转换符号变为取反加一，并规定0取反加一得到最小负数。这使得计算时只需完成加法运算，同时无需区分符号位。

   而本题中，由于最小负数取反加一的运算带来了符号位的两次变化，导致溢出。为避免这一问题，只能修改底层中数字计算方法（如计算时将符号位区分开），而这样会影响计算速度，由于使用时大数可以用浮点数正确表示，因此这样极端的情况很少出现，考虑到计算速度，不值得更改底层设定。

4. x--并输出，得到结果为：

   ```
   2147483647
   ```

   再将x++并输出，得到：

   ```
   -2147483648
   ```

   可知x++，x--得到的答案验证了之前的规定，并且实现的是加1与加-1。

问题二：

系统里以白特(byte)为寻址单位，写个C语言程序，说明一个4白特的int：0x12345678在机器里是怎么存放的？

解答：

测试程序：

```
#include <iostream>
#include <stdio.h>
int main()
{
    int a = 0x12345678;
    printf("%d\n", a);
    char *c = (char*)&a;
    printf("%x %x %x %x", c[0], c[1], c[2], c[3]);
```

```
    return 0;
}
```

输出：

```
305419896
78 56 34 12
```

可知数据的高位字节保存在内存的高地址中，使得地址的高低和数据位权有效地结合，为小端模式。

同时可以将字符串序列按序存入整数中，并输出其代表的正数。

测试程序：

```
#include <iostream>
#include <stdio.h>
int main()
{
    char c[4] = {0x12, 0x34, 0x56, 0x78};
    int *a = (int *)c;
    printf("%x", *a);
    return 0;
}
```

输出：

```
78563412
```

这也验证了存储模式为小端模式。

体会：

本周的作业在之前学习c语言程序设计基础时有过涉及，当时因为缺乏硬件学习基础，没能很好的理解原码、反码和补码。这次的作业我首先重新学习了相关知识，并且验证了越界的情况，以及思考在设计中如何避免这类情况，同时对计算机研发的思想过程和相对取舍有了更深的理解。