**第 2 章 信息的表示和处理**

第二章主要是在说明现代机器及C语言如何编码整数值和浮点数值，也就是在机器中如何表示整数信息、浮点数信息。

**我会在此处记录第二章的家庭练习答案和一些课后习题的思路，不过重要的不是结果，而是思考的过程。**

* 首先要说明的是，我的机器在 C 语言中是用补码表示有符号数，并且我的机器是 64 位字长的机器。
* C 语言标准并未规定人们应该用何种编码表示有符号数，不过现代机器大都用补码表示有符号数，或许某天我们会遇见一种更加完美的编码方式！
* 两个数的 w 位补码之和与无符号之和有完全相同的位级表示
* 两个数的 w 位补码乘和无符号乘也有完全相同的位级表示
* 有一个很灵魂的问题，C 语言中显式的强制类型转换与隐式的类型转换之间有什么区别？
* 有时也会好奇 sizeof 函数会返回什么，我在此处找到了[答案](https://en.cppreference.com/w/c/language/sizeof)，会返回一个无符号的 size\_t！

**课后习题参考地址**

* [dreamanddead.github.io](https://dreamanddead.github.io/CSAPP-3e-Solutions/chapter2/2.62/)

**2.34**

将位表示按照无符号和补码方式编码即可。

* 注意补码乘法的操作规则：首先按照正常的乘法计算乘积，之后将乘积模上 2^w，最后完成无符号数到补码表示的转换。

**2.35**

* 好题！

其实这里就需要一些数论基础了，不然看起来会很难受。好在书中给出了详细的分析过程，推荐一定自己写一遍。

**2.44**

**A.**

对任意的 int 数据 x 来说，如果 x 大于 0，那么我们可以知道 x 大于等于 1（整数）。那么 x-1 一定不会小于 0

反之，如果 x 小于等于 0，那么 x 必定小于等于 -1（整数），那么 x-1 必定小于等于 -1，此时 x-1 < 0 为真。

**也就是说，x > 0 和 x-1 < 0 是互斥的，所以 x >0 || x-1 <0 是永真的吗？**

从逻辑上来说是的，但是 C 语言只能以有限的位长表示有符号整数，本题是 32 位的 int。

那么如果 x 为 也就是 ，此时 x-1 会发生负溢出，结果将变成 ，这显然是一个正数，在这种情况下，我们有 x > 0 为假，且 x-1 < 0 也为假，从而可以知道整个表达式为假。

**B.**

假设 为假，那么 x 的最低三位就为 111，所以 x<<29 位后的位表示应为 111 + 29 个 0。根据补码编码，该位表示对应的数值为 -536870912，所以 (x<<29) < 0 为真，整体表达式为真。

**C.**

逻辑上来看显然一个平方数大于等于 0 恒成立，可惜现在我们用 int 来表示这些数，答案或许就不那么肯定了。

该表达式是在做一个补码的乘法，根据书中知识我们知道补码乘法是先计算这两个数的结果然后将其模上 ，之后再将无符号数转成有符号数。

若是取模后的结果为 ，该数超过了 32 位补码表示最大值，所以会发生溢出。那么最终的结果应是 ，这是一个负数，所以表达式为假了。

**D.**

真，参考 66 页的补码非运算即可

**E.**

假，如果 x 为 ，那么 x = -x，此时两者都为负数，表达式为假。

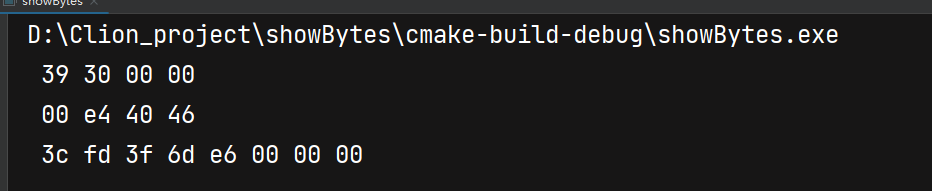
**F.**

真，注意 C 语言在对有符号数和无符号进行比较时，会将有符号数转换成无符号数。

**G.**

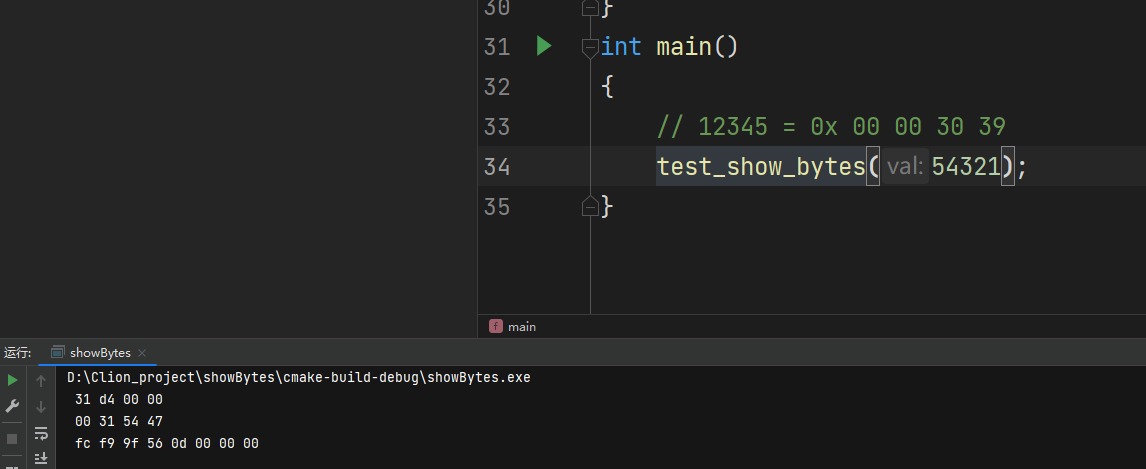
**2.55**

|  |
| --- |
| C // // Created by mirror on 2023/4/28. // #include <stdio.h> typedef unsigned char \*byte\_pointer; void show\_bytes(byte\_pointer start, size\_t len){  size\_t i;  for(i = 0; i < len; i++)  printf(" %.2x", start[i]);  printf("\n");  } void show\_int(int x){  show\_bytes((byte\_pointer) &x,sizeof(int)); } void show\_float(float x){  show\_bytes((byte\_pointer) &x, sizeof(float )); } void show\_pointer(void \*x){  show\_bytes((byte\_pointer) &x, sizeof(void \*)); } void test\_show\_bytes(int val){  int ival = val;  float fval = (float) ival;  int \*pval = &ival;  show\_int(ival);  show\_float(fval);  show\_pointer(pval);  } int main() {  // 12345 = 0x 00 00 30 39  test\_show\_bytes(12345); } |



喔，我是小端法的机器耶，因为最低有效字节在前面。同时可以发现在我的机器上 int 和 float 都是用 4 个字节来存储的。然后用 8 个字节存储 void \* 类型的指针。

**2.56**



**2.57**

**2.59**

主要涉及到两个问题：

1. 如何获取 x 的最低有效字节。
2. 如何将 y 的最低有效字节置零，同时保持 y 的其他字节不变。

下面是我的代码

|  |
| --- |
| SQL int test255(int x,int y) {  return (x & 0xFF) | (y & (~(0xFF))); } |

|  |
| --- |
| SQL int main() {  printf("%d\n",sizeof(int));  int x = 0x89abcdef; int y = 0x765432EF;  printf("%x\n",test255(x,y));  } |

**2.60**

这个题很多细节，但是整体不难。

我的思路如下：

1. 首先得到 x 的低 i 个字节的值，不妨记为 y，这可以通过与运算得到。但我们需要有 2i 个 F。
2. 将 x 算术右移 i 个字节，也就是 8i 位
3. 将 x 的最低有效字节置零
4. 将 x 或上 b
5. 将 x 左移 i 个字节
6. 将 x 或上 y
7. 返回 x

程序：

|  |
| --- |
| C void show\_unsigned(unsigned x){  show\_bytes((byte\_pointer) &x, sizeof(unsigned)); } unsigned replace\_byte(unsigned x,int i,unsigned char b){  /\*\*  \* 这里 i 是有符号数，如果小于零的话，程序如何处理？  \* 我暂时直接返回原数  \*/  if (i < 0){  printf("i 小于零\n");  return x;  }  unsigned s = (1<<(8\*i))-1; // 一个隐式的类型转换，从 int 转 unsigned，但是对结果没有影响  show\_unsigned(s);  unsigned y = x & s;  show\_unsigned(y);   x = x >> (8\*i);  show\_unsigned(x);   x = x & (~0xff);  show\_unsigned(x);   x = x | b;  show\_unsigned(x);   x = x << (8\*i);  show\_unsigned(x);   x |= y;   show\_unsigned(x);  return x;   }   int main() {   replace\_byte(0x12345678,2,0xab);   } |

**2.61**

很简单的题目，只需要用位运算和逻辑运算即可完成，并且注意到 A 和 B 在逻辑上是相反的，C、D 同理。

|  |
| --- |
| C int get\_msb(int x);  int msb\_all\_one(int x) {  int tmp = get\_msb(x);  return all\_one(tmp); }  int msb\_all\_zero(int x) {  return !(msb\_all\_one(x));  }  int get\_msb(int x) {  int shift\_val = (sizeof(int) - 1) << 3;  int xright = x >> shift\_val;  return xright & 0xFF;  }  int main() {   int test = ~0;  int test1 = 0;   printf("%d\n", all\_one(test));   printf("%d\n", all\_zero(test1));   } |

**2.62**

* 目标：如果机器对 int 数进行算术右移返回 1，其他情况返回 0。代码应该可以运行在任何字长的机器上。

~~好难想，右移一共有几种呢？书上给出了逻辑和算术两种，那我姑且认为有两种吧。~~

* ~~如何判断一个 int 数进行的是算术右移还是逻辑右移呢？~~

~~按照题目的意思， int 是用补码表示的，那么如果是一个负数的 int，其最高位作为符号位应为 1。~~

~~此时假设我们右移一位的话，那么逻辑右移会得到一个正值，而算术右移值仍然为负。~~

~~能否通过判断移位后的值的正负来决定是算术右移还是逻辑右移呢？~~

~~如果可以的话，按照这个思路，我们应该去判断移位后的结果的正负值，也就是要判断一个数是正是负。如果是正，必然是逻辑右移。如果是负，则是算术右移。~~

~~如何判断一个值的正负？由于是补码表示的，所以我们可以通过符号位来判断。符号位为1，表示是负，否则为正。~~

~~符号位就是最高 bit 位，那么也就是要获取到 int 数的最高 bit 位，如何获取？~~

* ~~我们可以先拿到最高有效字节，然后与上 0x80 即可获得最高 bit 位。~~

|  |
| --- |
| C int get\_msb(int x) {   int shift\_val = (sizeof(int) - 1) << 3;  int xright = x >> shift\_val;  return xright & 0xFF;  }  int int\_shifts\_are\_arithmetic(){  int test = -6; // 1101  int ans = test >> 1;   int msb = get\_msb(ans);   int msb\_bit = msb & 0x80;   if (msb\_bit)  {  return 1; //是算术右移  }    } |

* 看了别人的实现后，发现可以利用 -1 的位表示为 全 1 的特性来判断是算术右移还是逻辑右移，具体有如下代码：

|  |
| --- |
| C /\*  \* int-shifts-are-arithemetic.c  \*/  #include <stdio.h> #include <assert.h>  int int\_shifts\_are\_arithemetic() {  int num = -1;  return !(num ^ (num >> 1)); }  int main(int argc, char\* argv[]) {  assert(int\_shifts\_are\_arithemetic());  return 0; } |

如果是算术右移，那么右移 1 位之后保持不变，异或之后为 0，再取逻辑非运算返回 1.

如果是逻辑右移，那么右移 1 位之后最高位为 0，异或之后不为 0，再取逻辑非返回 0.