ICS Homework Week 5

October 8, 2019

I. Byte Order

1. Consider the following function [`sizeof(int) == 4`]

```
typedef unsigned char byte;
void show_bytes(byte *start, int len) {
      int i;
      for(i = 0; i < len; i++)
          printf("%.2x", start[i]);
}
unsigned int val = 0x654321;
byte *valp = (byte *) &val;
```

(1) What is the value of valp[] (if consider it as an array of 4 elements of "`byte`")?

Little-endian: {valp[0], valp[1], valp[2], valp[3]} = {0x21, 0x43, 0x65, 0x00}

Big-endian: {valp[0], valp[1], valp[2], valp[3]} = {0x00, 0x65, 0x43, 0x21}

(2) What is the output of the following call to show_bytes on big-endian and little-endian machines respectively? (You can have a try on your machine!)

	Little-endian	Big-endian
show_bytes(valp, 1)	21	00
show_bytes(valp, 2)	2143	0065
show_bytes(valp, 4)	21436500	00654321

2. Write a procedure `is_little_endian` that will return 1 when compiled and run on a little-endian machine, and will return 0 when compiled and run on a big-endian machine. This program should run on any machine, regardless of its word.

```
int is_little_endian(void) {
    // fill in C codes ...
    // not the only answer
    int x = 1;
    return !(1 - (int)(*(char *)&x));
}
```

3. Write a procedure `to_little_endian` that will transform an 32 bit unsigned. integer number into little endian format and return the pointer to the little endian integer. This program should run on both little endian and big endian machine. (Hint: you can use `is_little_endian` function written before)

```
byte *to_little_endian(unsigned int num) {
      // fill in C codes ...
      // not the only answer
      byte *ret = (byte *)malloc(sizeof(int));
      if (is_little_endian()) {
          memcpy(ret, &num, sizeof(int));
      } else {
          byte *bytes = (byte *)&num;
          ret[0] = bytes[3];
          ret[1] = bytes[2];
          ret[2] = bytes[1];
          ret[3] = bytes[0];
     }
     return ret;
}
```

II. Integer Size

You are given the task of writing a procedure `int_size_is_32` that yields 1 when run on a machine for which an int is 32 bits, and yields 0 otherwise. You are not allowed to use the sizeof operator. Here is a first attempt:

```
int bad_int_size_is_32() {

   int set_msb = 1 << 31;
   int beyond_msb = 1 << 32;

   return set_msb && !beyond_msb;
}
```

However, when compiled and run on a 32-bit SUN SPARC, this procedure returns 0. The following compiler message gives us an indication of the problem:

```
warning: left shift count >= width of type
```

1. In what way does our code fail to comply with the C standard? (Hint: you can search online for the C language standard on shift operations)

"If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined."

(http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf 6.5.7)

2. Modify the code to run properly on any machine for which data type int is at least 32 bits.

```
int int_size_is_32() {
    // fill in C codes
    int set_msb = 1 << 31;
    int beyond_msb = set_msb << 1;
    // or
    // int beyond_msb = 2 << 31;
    return set_msb && !beyond_msb;
}
```

3. Modify the code to run properly on any machine for which data type int is at least 16 bits.

```
int int_size_is_32_16bit() {
    // fill in C codes
    int set_msb = 1 << 15 << 15 << 1;
    int beyond_msb = set_msb << 1;

    return set_msb && !beyond_msb;
}
```