CS 61C Spring 2024 — C and Number Representation — Discussion 1

1 Pre-Check: Introduction to C

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 The correct way of declaring a character array is `char[] array`.

1.2 True or False: C is a pass-by-value language.

2 Pass-by-who?

2.1 The following functions may contain logic or syntax errors. Find and correct them.

(a) Returns the sum of all the elements in `summands`.

```
int sum(int *summands) {
    int sum = 0;
    for (int i = 0; i < sizeof(summands); i++)
        sum += *(summands + i);
    return sum;
}
```

(b) Copies the string `src` to `dst`.

```
void copy(char *src, char *dst) {
    while (*dst++ = *src++);
}
```

(c) Increments all of the letters in the `string` which is stored at the front of an array of arbitrary length, `n >= strlen(string)`. Does not modify any other parts of the array's memory.

```
void increment(char *string, int n) {
    for (int i = 0; i < n; i++)
        *(string + i)++;
}
```

(d) Overwrites an input string `src` with "61C is awesome!" if there's room. Does nothing if there is not. Assume that `length` correctly represents the length of `src`.

```
1  void cs61c(char *src, size_t length) {
2      char *srcptr, replaceptr;
3      char replacement[16] = "61C is awesome!";
4      srcptr = src;
5      replaceptr = replacement;
6      if (length >= 16) {
7          for (int i = 0; i < 16; i++)
8              *srcptr++ = *replaceptr++;
9      }
10 }
```

2.2 Implement the following functions so that they work as described.

(a) Swap the value of two **int**s. *Remain swapped after returning from this function.* Hint: Our answer is around three lines long.

```
void swap(________________, ________________) {
```

(b) Return the number of bytes in a string. *Do not use* `strlen`. Hint: Our answer is around 5 lines long.

```
int mystrlen(________________) {
```

3 Pre-Check: Number Representation

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

3.1 Depending on the context, the same sequence of bits may represent different things.

3.2 It is possible to get an overflow error in Two's Complement when adding numbers of opposite signs.

3.3 If you interpret a N bit Two's complement number as an unsigned number, negative numbers would be smaller than positive numbers.

3.4 If you interpret an N bit Bias notation number as an unsigned number (assume there are negative numbers for the given bias), negative numbers would be smaller than positive numbers.

3.5 We can represent fractions and decimals in our given number representation formats (unsigned, biased, and Two's Complement).

4 Unsigned and Signed Integers

4.1 If we have an n-digit unsigned numeral $d_{n-1}d_{n-2}\ldots d_0$ in *radix* (or *base*) r, then the value of that numeral is $\sum_{i=0}^{n-1} r^i d_i$, which is just fancy notation to say that instead of a 10's or 100's place we have an r's or r^2's place. For the three radices binary, decimal, and hex, we just let r be 2, 10, and 16, respectively.

Let's try this by hand.

(a) Convert the following numbers from their initial radix into the other two common radices:

1. 0b10010011
2. 0
3. 437
4. 0x0123

(b) Convert the following numbers from hex to binary:

1. 0xD3AD
2. 0x7EC4

4.2 Unsigned binary numbers work for natural numbers, but many calculations use negative numbers as well. To deal with this, a number of different schemes have been used to represent signed numbers. Here are two common schemes:

Two's Complement:

- We can write the value of an n-digit two's complement number as $\sum_{i=0}^{n-2} 2^i d_i - 2^{n-1} d_{n-1}$.
- Negative numbers will have a 1 as their most significant bit (MSB). Plugging in $d_{n-1} = 1$ to the formula above gets us $\sum_{i=0}^{n-2} 2^i d_i - 2^{n-1}$.
- Meanwhile, positive numbers will have a 0 as their MSB. Plugging in $d_{n-1} = 0$ gets us $\sum_{i=0}^{n-2} 2^i d_i$, which is very similar to unsigned numbers.
- To negate a two's complement number: flip all the bits and add 1.
- Addition is exactly the same as with an unsigned number.
- Only one 0, and it's located at 0b0.

Biased Representation:

- The number line is shifted so that the smallest number we want to be representable would be 0b0...0.
- To find out what the represented number is, read the representation as if it was an unsigned number, then add the bias.
- We can shift to any arbitrary bias we want to suit our needs. To represent (nearly) as much negative numbers as positive, a commonly-used bias for N bits is $-(2^{N-1} - 1)$.

For questions (a) through (c), assume an 8-bit integer and answer each one for the case of an unsigned number, biased number with a bias of -127, and two's complement number. Indicate if it cannot be answered with a specific representation.

(a) What is the largest integer? What is the result of adding one to that number?

1. Unsigned?
2. Biased?
3. Two's Complement?

(b) How would you represent the numbers 0, 1, and -1?

1. Unsigned?
2. Biased?
3. Two's Complement?

(c) How would you represent 17 and -17?

1. Unsigned?
2. Biased?
3. Two's Complement?

4.3 Prove that the two's complement inversion trick is valid (i.e. that x and $\overline{x}+1$ sum to 0).

5 Arithmetic and Counting

5.1 Addition and subtraction of binary/hex numbers can be done in a similar fashion as with decimal digits by working right to left and carrying over extra digits to the next place. However, sometimes this may result in an overflow if the number of bits can no longer represent the true sum. Overflow occurs if and only if two numbers with the same sign are added and the result has the opposite sign.

(a) Compute the decimal result of the following arithmetic expressions involving 6-bit Two's Complement numbers as they would be calculated on a computer. Do any of these result in an overflow? Are all these operations possible?

1. 0b011001 − 0b000111
2. 0b100011 + 0b111010
3. 0x3B + 0x06
4. 0xFF − 0xAA
5. 0b000100 − 0b001000

(b) How many distinct numbers can the following schemes represent? How many distinct *positive* numbers?

1. 10-bit unsigned
2. 8-bit Two's Complement
3. 6-bit biased, with a bias of -30
4. 10-bit sign-magnitude

6 Endianness

- Machines are byte-addressable. Memory is like a large array of cells. Each storage cell stores 8 bits, and these byte cells are ordered with an address.
- A 32b architecture has 32 bit memory addresses, addresses 0x00000000 - 0xFFFFFFFF

Typed variables

- Examples: int, long, char
- sizeof(dataType) indicates the number of bytes in memory required to store a particular data type

Pointers

- a variable whose value is an address of another variable
- Declaration: dataType* name;
- Dereference operator: Based on the pointer declaration statement, the compiler fetches the corresponding amount of bytes. For example, if p is a pointer to a 4 byte integer variable x, then *p involves fetching 4 bytes starting from the address of x, which is the value of p. Therefore, the value of x and value of *p are equal

Endianness

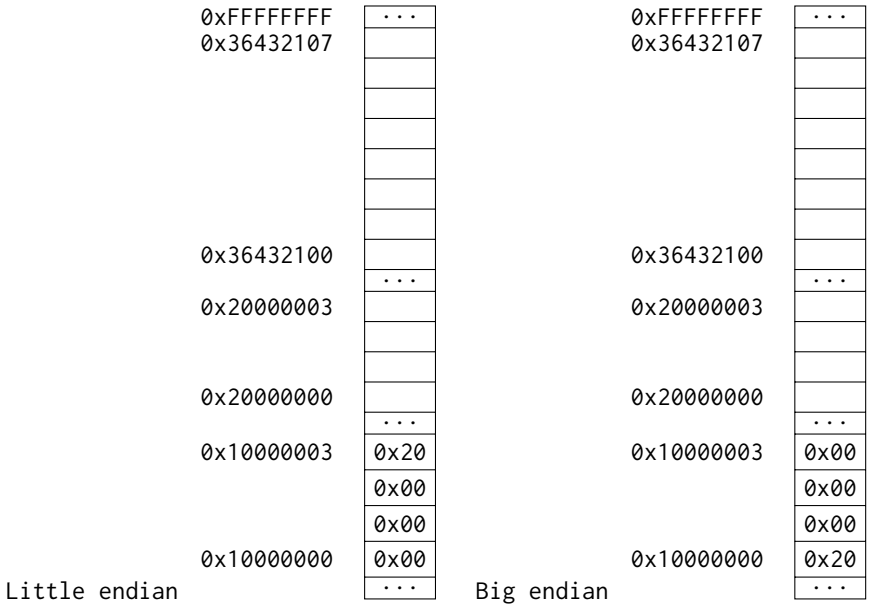
- Recall different data types are stored in x amount of contiguous byte cells in memory
- Big endian: the most significant byte of the value of a variable is stored in memory at the lowest address of the chunk of byte cells allocated for that variable
- Little endian: the least significant byte of the value of a variable is stored in memory at the lowest address of the chunk of byte cells allocated for the variable

6.1 Based on the following code and a 32b architecture, fill in the values located in memory at the byte cells for both a big endian and little endian system.

Suppose:

- the array nums starts at address 0x36432100
- p's address is 0x10000000

```
1 uint32_t nums[2] = {10, 20};
2 uint32_t* q = (uint32_t*) nums;
3 uint32_t** p = &q;
```



6.2 Provide two answers for the following questions: big endian system and little endian system

Suppose `uint64_t*` y = (`uint64_t*`) nums is executed after the code

1. What does *y evaluate to?

2. What does `&q` evaluate to? What does `&nums` evaluate to?

3. What does *(q+1) evaluate to?