CS 61C C and Number Representation
Spring 2024 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`.

False. The correct way is `char array[]`.

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

True. If you want to pass a reference to anything, you should use a pointer.

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`.

It is necessary to pass a size alongside the pointer.

```c
int sum(int* summands, size_t n) {
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += *(summands + i);
    return sum;
}
```

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

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

No errors.

(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.

The ends of strings are denoted by the null terminator rather than n. Simply having space for n characters in the array does not mean the string stored inside is also of length n.

```c
void increment(char* string) {
    for (i = 0; string[i] != 0; i++)
```

```
        string[i]++; // or (*(string + i))++;
}
```

Another common bug to watch out for is the corner case that occurs when incrementing the character with the value 0xFF. Adding 1 to 0xFF will overflow back to 0, producing a null terminator and unintentionally shortening the string.

(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`.

```
void cs61c(char *src, size_t length) {
    char *srcptr, replaceptr;
    char replacement[16] = "61C is awesome!";
    srcptr = src;
    replaceptr = replacement;
    if (length >= 16) {
        for (int i = 0; i < 16; i++)
            *srcptr++ = *replaceptr++;
    }
}
```

`char *srcptr, replaceptr` initializes a **`char`** pointer, and a **`char`**—not two **`char`** pointers.

The correct initialization should be, `char *srcptr, *replaceptr`.

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(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

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

```
int mystrlen(char* str) {
    int count = 0;
    while (*str != 0) {
        str++;
        count++;
    }
    return count;
}
```

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.

True. The same bits can be interpreted in many different ways with the exact same bits! The bits can represent anything from an unsigned number to a signed number or even, as we will cover later, a program. It is all dependent on its agreed upon interpretation.

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

False. Overflow errors only occur when the correct result of the addition falls outside the range of $[-(2^{n-1}), 2^{n-1} - 1]$. Adding numbers of opposite signs will not result in numbers outside of this range.

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

False. In Two's Complement, the MSB is always 1 for a negative number. This means EVERY negative number in Two's Complement, when converted to unsigned, will be larger than the 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.

True. In bias notation, we add a bias to the unsigned interpretation to create the value. Regardless of where we 'shift' the range of representable values, the negative numbers, when converted to unsigned, will always stay smaller than the positive numbers. This is unlike Two's Complement (see description above).

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

False. Our current representation formats has a major limitation; we can only represent and do arithmetic with integers. To successfully represent fractional values as well as numbers with extremely high magnitude beyond our current boundaries, we need another representation format.

4 Unsigned and Signed Integers

4.1 If we have an n-digit unsigned numeral $d_{n-1}d_{n-2}\dots d_0$ 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 = 147 = 0x93
2. 0 = 0b0 = 0x0
3. 437 = 0b0001 1011 0101 = 0x1B5
4. 0x0123 = 0b0000 0001 0010 0011 = 291

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

1. 0xD3AD = 0b1101 0011 1010 1101 = 54189
2. 0x7EC4 = 0b0111 1110 1100 0100 = 32452

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? 255, 0
2. Biased? 128, -127
3. Two's Complement? 127, -128

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

1. Unsigned? 0b0000 0000, 0b0000 0001, not possible
2. Biased? 0b0111 1111, 0b1000 0000, 0b0111 1110
3. Two's Complement? 0b0000 0000, 0b0000 0001, 0b1111 1111

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

1. Unsigned? 0b0001 0001, not possible
2. Biased? 0b1001 0000, 0b0110 1110
3. Two's Complement? 0b0001 0001, 0b1110 1111

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

Note that for any x we have $x+\overline{x} = 0b1\ldots1$. Adding 0b1 to 0b1...1 will cause the value to overflow, meaning that $0b1\ldots1 + 0b1 = 0b0 = 0$. Therefore, $x+\overline{x}+1 = 0$

A straightforward hand calculation shows that $0b1\ldots1 + 0b1 = 0$.

5 Arithmetic and Counting

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.

5.1 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?

(a) $0b011001 - 0b000111$

$0b010010 = 18$, No overflow.

(b) $0b100011 + 0b111010$

Adding together we get 0b1011101, however since we are working with 6-bit numbers we truncate the first digit to get $0b011101 = 29$. Since we added two negative numbers and ended up with a positive number, this results in an overflow.

(c) 0x3B + 0x06

Converting to binary, we get 0b111011 + 0b000110 = (after truncating as the problem states we're working with 6-bit numbers) 0b000001 = 1. Despite the extra truncated bit, this is not an overflow as -5 + 6 indeed equals 1!

(d) 0xFF − 0xAA

Trick question! This is not possible, as these hex numbers would need 8 bits to represent and we are working with 6 bit numbers.

(e) 0b000100 − 0b001000

The 2's complement of 0b001000 is 0b110111 + 1 = 0b111000. We add that to 0b000100 to get 0b111100.

We can logically fact check this by converting everything to decimals: 0b000100 is 4 and 0b001000 is 8, so the subtraction should result in -4, which is 0b111100.

5.2 How many distinct numbers can the following schemes represent? How many distinct *positive* numbers?

(a) 10-bit unsigned 1024, 1023

In unsigned representation, different bit-strings correspond to different numbers, so 10 bits can represent $2^{10} = 1024$ distinct numbers. Out of all of these, only the number 0 is non-positive, so we can represent 1023 distinct positive numbers.

(b) 8-bit Two's Complement 256, 127

Like unsigned, different bit-strings correspond to distinct numbers in Two's Complement, so 8 bits can represent $2^8 = 256$ numbers. Out of these, half of them have a MSB of 1, which are negative numbers, and one is the number zero, so we can represent $256/2 - 1 = 127$ distinct positive numbers.

(c) 6-bit biased, with a bias of -30 64, 33

Also like unsigned, in biased notation, no two different bit-strings correspond to the same number, so 6 bits can represent $2^6 = 64$ numbers. With this bias, the largest number we can represent is 0b111111$= 63 - 30 = 33$, and the smallest is -30, so there are 33 distinct positive numbers ($1 \sim 33$).

(d) 10-bit sign-magnitude 1023, 511

Two different bit-strings (0b0000000000 and 0b1000000000) correspond to the same number zero, so we can represent only $2^{10} - 1 = 1023$ distinct numbers. Out of these, every bit-string with a MSB of 0, except 0b0000000000, correspond to a different positive number, so we can represent $2^9 - 1 = 511$ distinct positive numbers.

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

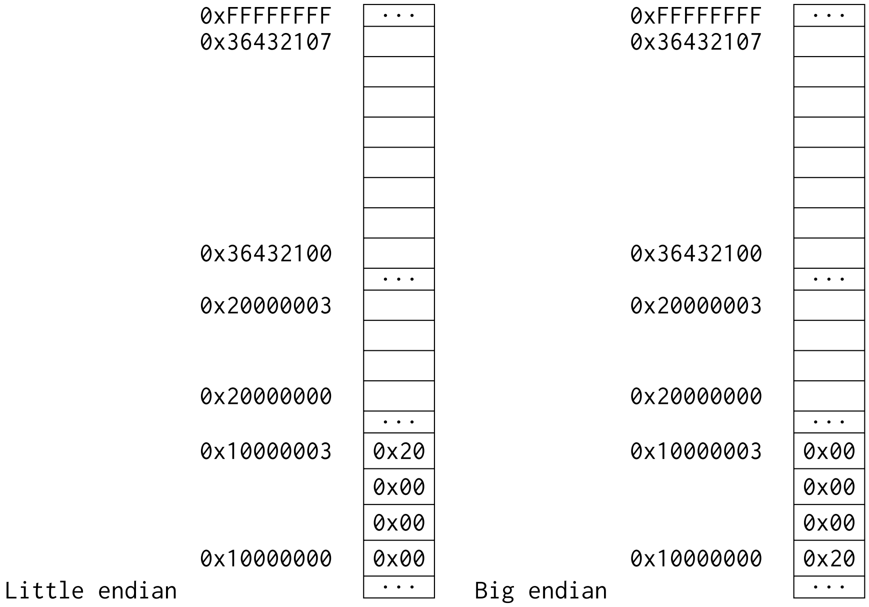
- Recall different data types are stored in x amount of contiguous byte cells in memory
- Big endian: the most significant byte of the value of a variable is stored in memory at the lowest address of the chunk of byte cells allocated for that variable
- Little endian: the least significant byte of the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

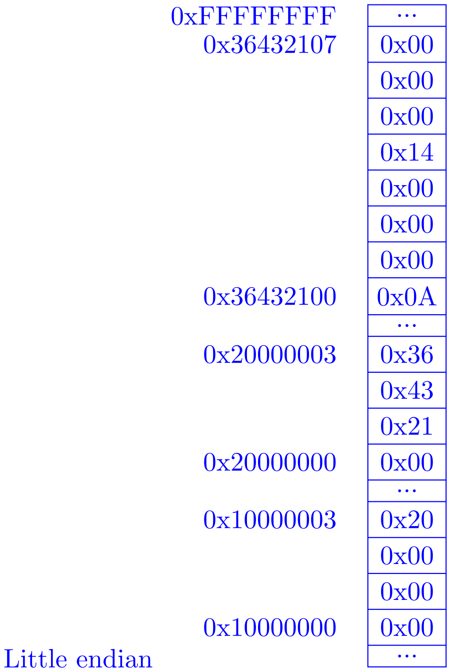
6.1 Based on the following code and a 32b architecture, fill in the values located in memory at the byte cells for both a big endian and little endian system.

Suppose:

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

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


0xFFFFFFFF
0x36432107
0x36432100
0x20000003
0x20000000
0x10000003
0x20
0x00
0x00
0x10000000
0x00
Little endian
0xFFFFFFFF
0x36432107
0x36432100
0x20000003
0x20000000
0x10000003
0x00
0x00
0x00
0x10000000
0x20
Big endian


0xFFFFFFFF
0x36432107
0x00
0x00
0x00
0x14
0x00
0x00
0x00
0x36432100
0x0A
0x20000003
0x36
0x43
0x21
0x20000000
0x00
0x10000003
0x20
0x00
0x00
0x10000000
0x00
Little endian

0xFFFFFFFF	···
0x36432107	0x14
	0x00
	0x00
	0x00
	0x0A
	0x00
	0x00
0x36432100	0x00
	···
0x20000003	0x00
	0x21
	0x43
0x20000000	0x36
	···
0x10000003	0x00
	0x00
	0x00
0x10000000	0x20
	···

Big endian

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?

   Because y is a pointer to a `uint64_t`* variable, dereferencing results in evaluating 8 contiguous bytes starting from the value of y (an address in memory = 0x36432100) in big endian or little endian.

   Little-endian: 0x00000014 0000000A

   Big-endian: 0x0000000A 00000014

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

   `&q` evaluates to 0x20000000 in both big endian and little endian. This is the value of variable p (p is located at 0x10000000). `&nums` evaluates to 0x36432100.

   Both q and nums act as pointers to the first element of the nums array. However, nums is not like a variable. The values of `nums` and `&nums` are equal, while the address of variable q is not equal to the address of the data it is pointing to.

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

   *(q+1) = nums[1] = *(nums+1) = 20 (decimal). q and nums have the same value. q is a pointer to a 32 bit integer. Therefore, *(q+1) means the 4 bytes stored starting at address = q plus 1*sizeof(`uint32_t`) = 0x36432100 + 0x4 = 0x36432104 evaluated in big endian or little endian