CS61C Summer 2025 — C & Number Representation, Discussion 1

1 Precheck: Number Representation

1.1 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.

1.2 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 $\left[-\left(2^{\{n-1\}}\right), 2^{\{n-1\}} - 1\right]$. Adding numbers of opposite signs will not result in numbers outside of this range.

1.3 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.

1.4 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).

1.5 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.

2 Unsigned and Signed Integers

2.1 Convert the following numbers from their initial radix into the other two common radices:

(a) `0b10010011`

Decimal: `147`, Hex: `0x93`

(b) `0`

Binary: `0b0`, Hex: `0x0`

(c) `437`

Binary: `0b110110101`, Hex: `0x1B5`

(d) `0x0123`

Binary: `0b100100011`, Decimal: `291`

2.2 Convert the following numbers from hex to binary:

(a) `0xD3AD`

`0b1101001110101101`

(b) `0x7EC4`

`0b0111111011000100`

2.3 Assuming an 8-bit integer and a bias of −127 where applicable, what is the largest integer for each of the following representations? What is the result of adding one to that number?

(a) Unsigned

Largest: `255`, Largest + 1: `0`

(b) Biased

Largest: `128`, Largest + 1: `-127`

(c) Two's Complement

Largest: `127`, Largest + 1: `-128`

2.4 How would you represent the numbers 0, 1, and −1? Express your answer in binary and a bias of −127 where applicable.

(a) Unsigned

0: `0b0000 0000`, 1: `0b0000 0001`, −1: N/A

(b) Biased

0: `0b0111 1111`, 1: `0b1000 0000`, −1: `0b0111 1110`

(c) Two's Complement

0: `0b0000 0000`, 1: `0b0000 0001`, −1: `0b1111 1111`

2.5 How would you represent the numbers 17 and −17? Express your answer in binary and a bias of −127 where applicable.

(a) Unsigned

17: `0b0001 0001`, −17: N/A

(b) Biased

17: `0b1001 0000`, −17: `0b0110 1110`

(c) Two's Complement

17: `0b0001 0001`, −17: `0b1110 1111`

2.6 What is the largest integer that can be represented by *any* encoding scheme that only uses 8 bits?

There is no such integer. For example, an arbitrary 8-bit mapping could choose to represent the numbers from 1 to 256 instead of 0 to 255.

2.7 Prove that that $x + \overline{x} + 1 = 0$, where $\overline{x}$ is obtained by inverting the bits of x in binary.

Consider what happens when we perform $x + \overline{x}$. In each "place", we either have that x has a 0 bit in that place, meaning that $\overline{x}$ has a 1 bit in that place, or vice versa. In either case, adding `0b1` + `0b0` = `0b1`, meaning that regardless of the value of x, $x + \overline{x}$ = `0b111...111`. Adding 1 to this then causes overflow, resulting in 0.

3 Arithmetic and Counting

3.1 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`.

3.2 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 $\frac{256}{2} - 1 = 127$ 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 – 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.

4 Precheck: Introduction to C

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

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

4.2 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.

4.3 In compiled languages, the compile time is generally pretty fast, however the run-time is significantly slower than interpreted languages.

False. Reasonable compilation time, excellent run-time performance. It optimizes for a given processor type and operating system.

4.4 What is a pointer? What does it have in common with an array variable?

As we like to say, "everything is just bits." A pointer is just a sequence of bits, interpreted as a memory address. An array acts like a pointer to the first element in the allocated memory for that array. However, an array name is not a variable, that is, &arr = arr whereas &ptr != ptr unless some magic happens (what does that mean?).

4.5 If you try to dereference a variable that is not a pointer, what will happen? What about when you free one?

It will treat that variable's underlying bits as if they were a pointer and attempt to access the data there. C will allow you to do almost anything you want, though if you attempt to access an "illegal" memory address, it will segfault for reasons we will learn later in the course. It's why C is not considered "memory safe": you can shoot yourself in the foot if you're not careful. If you free a variable that either has been freed before or was not malloced/calloced/realloced, bad things happen. The behavior is undefined and terminates execution, resulting in an "invalid free" error.

4.6 Memory sectors are defined by the hardware, and cannot be altered.

False. The four major memory sectors, stack, heap, static/data, and text/code for any given process (application) are defined by the operating system and may differ depending on what kind of memory is needed for it to run.

What's an example of a process that might need significant stack space, but very little text, static, and heap space? (Almost any basic deep recursive scheme, since you're making many new function calls on top of each other without closing the previous ones, and thus, stack frames.)

What's an example of a text and static heavy process? (Perhaps a process that is incredibly complicated but has efficient stack usage and does not dynamically allocate memory.)

What's an example of a heap-heavy process? (Maybe if you're using a lot of dynamic memory that the user attempts to access.)

5 Pass-by-Who?

5.1 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.

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

(b) 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.

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

Additionally, the operator precedence is incorrect for the expression `*(string + i)++`. The `++` increment will occur *before* the dereference operator occurs. Thus, we need to wrap in parentheses with `(*(string + i))++` or use array subscripting–which has equal precedence as dereferencing–like `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.

(c) 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**.

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

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

5.2 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.

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