

1 Review: C Memory

1.1 For each part, choose one or more of the following memory segments where the data could be located: **code**, **static**, **heap**, **stack**.

(a) Static variables

Static

(b) Local variables

Stack

(c) Global variables

Static

(d) Constants (constant variables or values)

Code, static, or stack

Constants can be compiled directly into the code. `x = x + 1` can compile with the number 1 stored directly in the machine instruction in the code. That instruction will always increment the value of the variable `x` by 1, so it can be stored directly in the machine instruction without reference to other memory. This can also occur with pre-processor macros:

```
1 #define y 5
2
3 int plus_y(int x) {
4     x = x + y;
5     return x;
6 }
```

Constants can also be found in the stack or static storage depending on if it's declared in a function or not.

```
1 const int x = 1;
2
3 int sum(int* arr) {
4     int total = 0;
5     ...
6 }
```

In this example, `x` is a variable whose value will be stored in the static storage, while `total` is a local variable whose value will be stored on the stack. Variables declared **const** are not allowed to change, but the usage of **const** can get more tricky when combined with pointers.

(e) Functions (i.e. Machine Instructions)

Code

(f) Result of Dynamic Memory Allocation(`malloc` or `calloc`)

Heap

(g) String Literals

Static.

When declared in a function, string literals can only be stored in static memory. String literals are declared when a character pointer is assigned to a string declared within quotation marks, i.e. `char* s = "string"`. You'll often see a near identical alternative to declaring a string: `char s[7] = "string"`. This string array will be stored in the stack (when declared inside a function) and is mutable, though they cannot change in size. Note that the compiler will arrange for the char array to be initialised from the literal and be mutable.

Compare these different ways of storing "DEADBEEF". Assume that each program is run on the same machine and architecture.

```

1 char arr[] = "DEADBEEF"
1 int arr[2];
2 arr[0] = 0xDEADBEEF;
3 arr[1] = 0x00000000; //null terminator in hex is 0x00

```

1.2 Do these two C programs store "DEADBEEF" in memory the same way?

No, the first program stores "DEADBEEF" as a null-terminated array of chars, each element stored as their ASCII value (e.g D = 0x44), while the latter encodes the hexadecimal DEADBEEF as a 32b integer.

You take a look at the ASCII table and translate the string "DEADBEEF" into bytes.

```

1 int arr[2];
2 //storing "DEAD" in ascending order in arr[0]
3 arr[0] = 0x44454144
4
5 //storing "BEEF" in ascending order in arr[1]
6 arr[1] = 0x42454546

```

1.3 Does this C program store "DEADBEEF" in memory the same way as storing it as a string?

It depends. A string is stored with the leftmost character in the lowest address, up to the null terminator, regardless of the endianness. Storing the string as an integer changes depending on the system architecture. In a big-endian system, the integers would be stored in the same order byte-by-byte as the string "DEADBEEF". However, in a little-endian system, the Least Significant Byte is stored in the

lowest address, so `arr[0]` in memory would actually be stored, in ascending order, as `0x44 0x41 0x45 0x44`, and `arr[1]` as `0x46 0x45 0x45 0x42`.

2 Pre-Check: Floating Point

- 2.1 The idea of floating point is to use the ability to move the radix (decimal) point wherever to represent a large range of real numbers as exact as possible.

True. Floating point:

- Provides support for a wide range of values. (Both very small and very large)
- Helps programmers deal with errors in real arithmetic because floating point can represent $+\infty$, $-\infty$, NaN (Not a number)
- Keeps high precision. Recall that precision is a count of the number of bits in a computer word used to represent a value. IEEE 754 allocates a majority of bits for the significand, allowing for the use of a combination of negative powers of two to represent fractions.

- 2.2 Floating Point and Two's Complement can represent the same total amount of numbers (any reals, integer, etc.) given the same number of bits.

False. Floating Point can represent infinities as well as NaNs, so the total amount of representable numbers is lower than Two's Complement, where every bit combination maps to a unique integer value.

- 2.3 The distance between floating point numbers increases as the absolute value of the numbers increase.

True. The uneven spacing is due to the exponent representation of floating point numbers. There are a fixed number of bits in the significand. In IEEE 32 bit storage there are 23 bits for the significand, which means the LSB represents 2^{-23} times 2 to the exponent. For example, if the exponent is zero (after allowing for the offset) the difference between two neighboring floats will be 2^{-23} . If the exponent is 8, the difference between two neighboring floats will be 2^{-15} because the mantissa is multiplied by 2^8 . Limited precision makes binary floating-point numbers discontinuous; there are gaps between them.

- 2.4 Floating Point addition is associative.

False. Because of rounding errors, you can find Big and Small numbers such that: $(\text{Small} + \text{Big}) + \text{Big} \neq \text{Small} + (\text{Big} + \text{Big})$

FP approximates results because it only has 23 bits for Significand.

3 Floating Point

The IEEE 754 standard defines a binary representation for floating point values using three fields.

- The *sign* determines the sign of the number (0 for positive, 1 for negative).

- The *exponent* is in **biased notation**. For instance, the bias is -127 which comes from $-(2^{8-1} - 1)$ for single-precision floating point numbers.
- The *significand* or *mantissa* is akin to unsigned integers, but used to store a fraction instead of an integer.

The below table shows the bit breakdown for the single precision (32-bit) representation. The leftmost bit is the MSB and the rightmost bit is the LSB.

1	8	23
Sign	Exponent	Mantissa/Significand/Fraction

For normalized floats:

$$\text{Value} = (-1)^{\text{Sign}} * 2^{\text{Exp} + \text{Bias}} * 1.\text{significand}_2$$

For denormalized floats:

$$\text{Value} = (-1)^{\text{Sign}} * 2^{\text{Exp} + \text{Bias} + 1} * 0.\text{significand}_2$$

Exponent	Significand	Meaning
0	Anything	Denorm
1-254	Anything	Normal
255	0	Infinity
255	Nonzero	NaN

Note that in the above table, our exponent has values from 0 to 255. When translating between binary and decimal floating point values, we must remember that there is a bias for the exponent.

3.1 Convert the following single-precision floating point numbers from hexadecimal to decimal or from decimal to hexadecimal. You may leave your answer as an expression.

- 0x00000000 0x421E4000
- 0 • 0xFF94BEEF
- 8.25 NaN
- 0x41040000 • $-\infty$
- 0x0000F00 0xFF800000
- $(2^{-12} + 2^{-13} + 2^{-14} + 2^{-15}) * 2^{-126}$ • $1/3$
- 39.5625 N/A — Impossible to actually represent, we can only approximate it

We'll go more into depth with converting 8.25 and 0x0000F00. For the sake of brevity, the rest of the conversions can be done using the same process.

To convert 8.25 into binary, we first split up our 32b hexadecimal number into three parts. The sign is positive, so our sign bit -1^S will be 0. Then, we can solve for our significand. We know that our number will have a non-zero exponent, so we will have a leading 1 for our mantissa. Splitting 8.25 into its integer and decimal portions, we can determine that 8 will be encoded in binary as 1000. and 0.25 will be .01 (the 1 corresponds to the 2^{-2} place), so by implying the MSB, our significand

will be 00001000.. Finally, we can solve for the exponent. As our leading 1 is in the 2^3 place to encode 8, we must use the bias in reverse to find what exponent we encode in binary. 130 added with a bias of -127 results in 3, so our exponent is 0b10000010. Our final binary number concatenated is 0 100 0001 0 000 0100 0000 0000 0000 0000, or 0x41040000.

For 0x00000F00, splitting up the hexadecimal gives us a sign bit and exponent bit of 0, and a significand of 0b 000 0000 0000 1111 0000 0000. We now know that this will be some sort of denormalized positive number. We can find out the true exponent by adding the bias + 1 to get the actual exponent of -126 . Then, we can evaluate the mantissa by inspecting the bits that are 1 to the right of the radix point, and finding the corresponding negative power of two. This results in the mantissa evaluated as $2^{-12} + 2^{-13} + 2^{-14} + 2^{-15}$. Combining these get the extremely small number $(-1)^0 * 2^{-126} * (2^{-12} + 2^{-13} + 2^{-14} + 2^{-15})$

4 More Floating Point Representation

As we saw above, not every number can be represented perfectly using floating point. For this question, we will only look at positive numbers.

- 4.1 What is the next smallest number larger than 2 that can be represented completely?

For this question, you increment the number by the smallest amount possible. This is the same as incrementing the significand by 1 at the rightmost location.

$$(1 + 2^{-23}) * 2 = 2 + 2^{-22}$$

- 4.2 What is the next smallest number larger than 4 that can be represented completely?

For this question, you increment the number by the smallest amount possible. This is the same as incrementing the significand by 1 at the rightmost location.

$$(1 + 2^{-23}) * 4 = 4 + 2^{-21}$$

- 4.3 What is the largest odd number that we can represent? Hint: At what power can we only represent even numbers?

To find the largest odd number we can represent, we want to find when odd numbers will stop appearing. This will be when the LSB will have a step size of 2, subtracted by 1. After this number, only even numbers can be represented in floating point.

We can think of each binary digit in the significant as corresponding to a different power of 2 to get to a final sum. For example, 0b1011 can be evaluated as $2^3 + 2^1 + 2^0$, where the MSB is the 3rd bit and corresponds to 2^3 and the LSB is the 0th bit at 2^0 .

We want our LSB to correspond to 2^1 , so by counting the number of mantissa bits (23) and including the implicit 1, we get a total exponent of 24. The smallest number

with this power would have a mantissa of 00..00, so after taking in account the implicit 1 and subtracting, this gives $2^{24} - 1$