CS61C
Fall 2025

Precheck: C
Discussion 2

1 Precheck: Introduction to C

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.

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

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

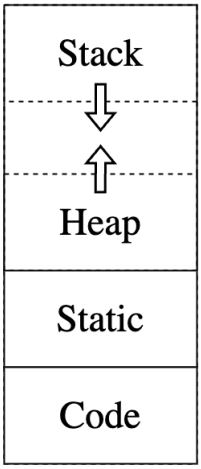
1.5 If you try to dereference a variable that is not a pointer, what will happen? What about when you free one?

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

2 Memory Management

C does not automatically handle memory for you, so it's up to you, the programmer, to allocate, use, and free memory correctly. In each program, an address space is set aside, separated into 2 dynamically changing regions and 2 'static' regions.

- **The Stack**: Stores local variables inside of functions. Data on the stack is garbage collected immediately after the *function in which it was defined* returns. Each function call creates a stack frame that holds the function's arguments and local variables. The stack grows downwards with nested function calls (LIFO structure), and shrinks upwards as functions return.
- **The Heap**: Stores memory manually allocated by the programmer with `malloc`, `calloc`, or `realloc`. Used for data that needs to persist after the function returns. Grows upwards in memory to 'meet' the stack. Memory on the heap is only freed when the programmer explicitly frees it. Careful heap management is necessary to avoid tricky bugs called Heisenbugs!
- **Static data**: Stores data that is a fixed size, like global variables and string literals. Does not grow or shrink through function execution.
- **Code (or Text)**: Is loaded at the start of the program and does not change after, contains executable instructions and any pre-processor macros.



There are a number of functions in C that can be used to dynamically allocate memory on the heap. The following are the ones we use in this class:

- `malloc(size_t size)` allocates a block of `size` bytes and returns the start of the block. The time it takes to search for a block is generally not dependent on `size`.
- `calloc(size_t count, size_t size)` allocates a block of `count * size` bytes, **sets every value in the block to zero**, then returns the start of the block.
- `realloc(void *ptr, size_t size)` "resizes" a previously-allocated block of memory to `size` bytes, returning the start of the resized block.
- `free(void *ptr)` deallocates a block of memory which starts at `ptr` that was previously allocated by the three previous functions.

3 Endianness

Machines are byte-addressable, meaning memory is organized as a large array of 1-byte cells, each with its own unique address. A 32b architecture uses 32-bit memory addresses, allowing the addresses to range from 0x00000000 to 0xFFFFFFFF.

Typed variables:

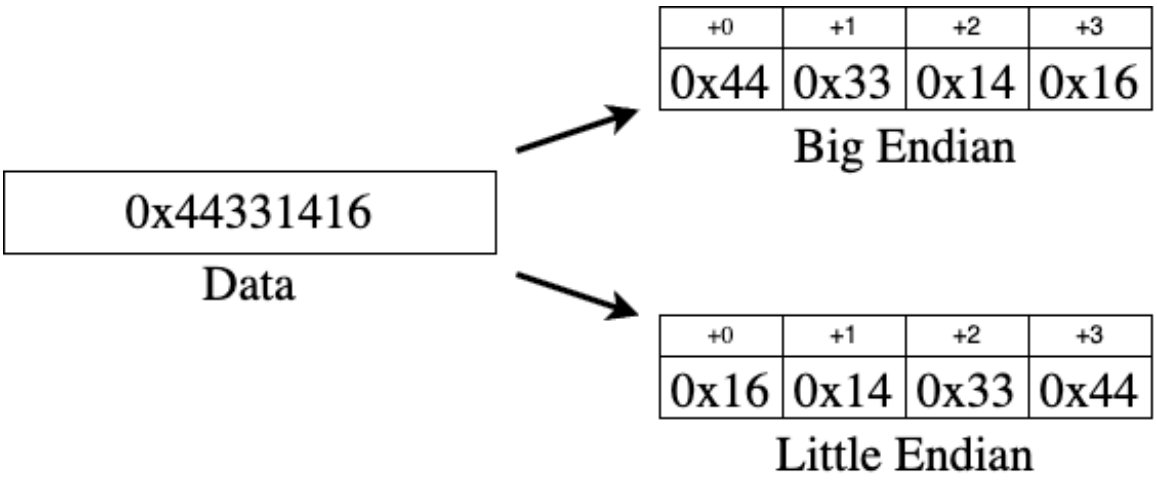
- Variables have a data type (ex: int, char, float) which determines the kind of data they can store
- `sizeof(dataType)` returns the number of bytes required to store a particular data type in memory

Pointers:

- A pointer is a variable that stores the memory address of another variable
- Declaration: `dataType* name;`
- Dereference operator (`*`): Fetches the data stored at the memory address the pointer points to. The pointer's type tells the compiler how many bytes to read. For instance, 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 the value of `*p` are equal.

Endianness:

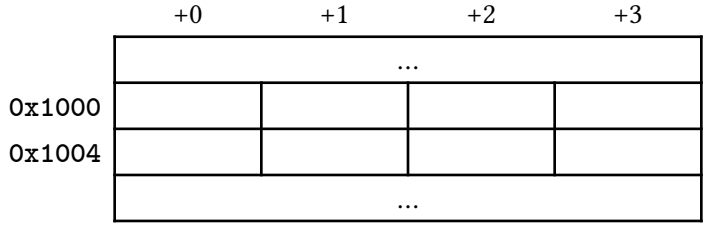
- Recall that each data type occupies a fixed number of contiguous bytes in memory. The order in which these bytes are stored depends on the system's endianness.
- **Big Endian:** the most significant byte of the value of a variable is stored in memory at the lowest address of the chunk of byte cells 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



3.1 Fill in the memory contents for each system after initializing **`arr`**. Assume **`arr`** begins at memory address `0x1000`.

```
uint32_t arr[2] = {0xD3ADB33F, 0x61C0FFEE};
```

(a) Little-Endian System



(b) Big-Endian System