CS61C Spring 2025

Precheck: C

Discussion 2

1 Precheck: Introduction to C

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.

1.3 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.

1.4 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?).

1.5 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.

1.6 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.)

2 Memory Management

C does not automatically handle memory for you. In each program, an address space is set aside, separated into 2 dynamically changing regions and 2 'static' regions.

- **The Stack**: local variables inside of functions, where data is garbage immediately after the *function in which it was defined* returns. Each function call creates a stack frame with its own arguments and local variables. The stack dynamically changes, growing downwards as multiple functions are called within each other (LIFO structure), and collapsing upwards as functions finish execution and return.
- **The Heap**: memory manually allocated by the programmer with `malloc`, `calloc`, or `realloc`. Used for data we want to persist beyond function calls, growing upwards to 'meet' the stack. Careful heap management is necessary to avoid Heisenbugs! Memory is freed only when the programmer explicitly frees it!
- **Static data**: global variables declared outside of functions, does not grow or shrink through function execution.
- **Code (or Text)**: 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. 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 pointer is 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 the value of `*p` are equal.

Endianness:
- Recall different data types are stored in a certain number 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

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

	+3	+2	+1	+0
	...			
0x1000	0xD3	0xAD	0xB3	0x3F
0x1004	0x61	0xC0	0xFF	0xEE
	...			

(b) Big-Endian System

	+3	+2	+1	+0
	...			
0x1000	0x3F	0xB3	0xAD	0xD3
0x1004	0xEE	0xFF	0xC0	0x61
	...			