CS 61C Summer 2022 — C and Memory Management — Discussion 1

1 Pre-Check

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 if false, correct the statement to make it true:

1.1 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.

1.2 The following is correct C syntax:

```
int num = 43
```

False. Semicolon!!

```
int num = 43;
```

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

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

1.5 Bitwise and logical operations result in the same behaviour for given bitstrings.

False. Bitwise and logical operations fundamentally speaking, perform the same operations, just in different contexts. Bitwise operations compare and operate on inputs bit-by-bit, from least to most significant bit in the bitstring. Logical operations compare and operate on inputs as a whole, where anything not 0 can be considered to be a 1.

Note that in 61C and both bitwise and logical operations, 0 can be considered as False and not-0 can be considered as True in comparisons!

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.)

1.7 When should you use the heap over the stack? Do they grow?

If you need to keep access to data over several function calls, use the heap. If you're dealing with a large piece of data, passing around a pointer to something on the heap is more efficient and a better practice than passing around the data itself. (Think: carrying a library around vs knowing the address TO the library). Heaps grow up and stacks grow down, meeting when working memory is full.

2 Memory Management

2.1 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

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.

```
const int x = 1;

int sum(int* arr) {
    int total = 0;
    ...
}
```

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) 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 (or data) memory. String literals are declared when a character pointer is assigned to a string declared within quotation marks, i.e. **`char*`** `s = "string"`. Strings declared this was are stored in the static memory segment. You'll often see 3 other ways in which to declare strings, one being very similar to string literals, as following: **`char`**`[7] s = "string"`. This is a string array will be stored in the stack and is mutable. Note that the compilier will arrange for the char array to be initialised from the literal and be mutable.

3 Bit-wise Operations

3.1 In C, we have a few bit-wise operators at our disposal:

- `AND` (`&`)
- `NOT` ($\sim$)
- `OR` (`|`)
- `XOR` ($\wedge$)
- `SHIFT LEFT` (`<<`)
  - Example: `0b0001 << 2 = 0b0100`
- `SHIFT RIGHT` (`>>`)
  - Example: `0b0100 >> 2 = 0b0001`

a	**b**	**a & b**	**a | b**	**a ∧ b**	**∼a**
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

For your convenience, truth tables for the logical operators are provided above. With the binary numbers `a`, `b`, and `c` below, perform the following bit-wise operations:

```
a = 0b1000 1011
b = 0b0011 0101
c = 0b1111 0000
```

(a) `a & b`

`0b0000 0001`

(b) `a ∧ c`

`0b0111 1011`

(c) `a | 0`

`0b1000 1011`
Anything `| 0` always evaluates to the original value, so this just returns the value of `a`.

(d) `a | (b >> 5)`

`0b1000 1011`
`b >> 5` evaluates to 0b0000 0001, so `a | 1` still returns the value of `a`.

(e) ∼ ((`b | c`) `& a`)

`0b0111 1110`