CS 61C Summer 2023 — C — Discussion 2

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 include an explanation:

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 What is a pointer? What does it have in common to 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?).

1.7 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.

1.8 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.)

1.9 For large recursive functions, you should store your data on the heap over the stack.

False. Generally speaking, if you need to keep access to data over several separate function calls, use the heap. However, recursive functions call themselves, creating multiple stack frames and using each of their return values. If you store data on the heap in a recursive scheme, your `malloc` calls may lead to you rapidly running out of memory, or can lead to memory leaks as you lose where you allocate memory as each stack frame collapses.

2 C

C is syntactically similar to Java, but there are a few key differences:

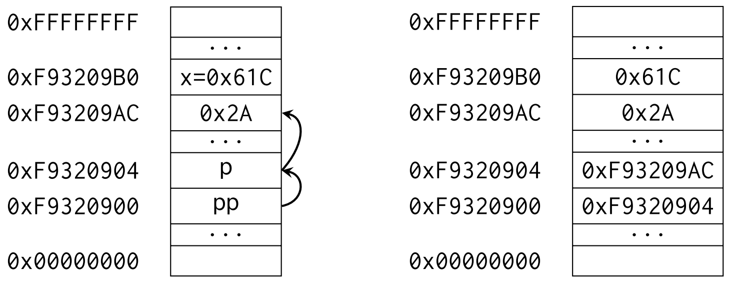
1. C is function-oriented, not object-oriented; there are no objects.
2. C does not automatically handle memory for you.

- Stack memory, or *things that are not manually allocated*: data is garbage immediately after the *function in which it was defined* returns.
- Heap memory, or *things allocated with* `malloc`, `calloc`, *or* `realloc`: data is freed only when the programmer explicitly frees it!
- There are two other sections of memory that we learn about in this course, *static* and *code*, but we'll get to those later.
- In any case, allocated memory always holds garbage until it is initialized!

3. C uses pointers explicitly. If `p` is a pointer, then `*p` tells us to use the value that `p` points to, rather than the value of `p`, and `&x` gives the address of `x` rather than the value of `x`.

   On the left is the memory represented as a box-and-pointer diagram.

   On the right, we see how the memory is really represented in the computer.



   Let's assume that **`int`**`* p` is located at `0xF9320904` and **`int`** `x` is located at `0xF93209B0`. As we can observe:

   - `*p` evaluates to `0x2A` (42_{10}).
   - `p` evaluates to `0xF93209AC`.
   - `x` evaluates to `0x61C`.
   - `&x` evaluates to `0xF93209B0`.

   Let's say we have an **`int`** `**pp` that is located at `0xF9320900`.

$\boxed{2.1}$ What does `pp` evaluate to? How about `*pp`? What about `**pp`?

`pp` evaluates to `0xF9320904`. `*pp` evaluates to `0xF93209AC`. `**pp` evaluates to `0x2A`.

$\boxed{2.2}$ The following functions are syntactically-correct C, but written in an incomprehensible style. Describe the behavior of each function in plain English.

(a) Recall that the ternary operator evaluates the condition before the ? and returns the value before the colon (:) if true, or the value after it if false.

```c
int foo(int *arr, size_t n) {
    return n ? arr[0] + foo(arr + 1, n - 1) : 0;
}
```

Returns the sum of the first N elements in `arr`.

(b) Recall that the negation operator, `!`, returns 0 if the value is non-zero, and 1 if the value is 0. The `~` operator performs a *bitwise not* (NOT) operation.

```c
int bar(int *arr, size_t n) {
    int sum = 0, i;
    for (i = n; i > 0; i--)
        sum += !arr[i - 1];
    return ~sum + 1;
}
```

Returns -1 times the number of zeroes in the first N elements of `arr`.

(c) Recall that `^` is the *bitwise exclusive-or* (XOR) operator.

```c
void baz(int x, int y) {
    x = x ^ y;
    y = x ^ y;
    x = x ^ y;
}
```

Ultimately does not change the value of either `x` or `y`.

(d) (Bonus: How do you write the *bitwise exclusive-nor* (XNOR) operator in C?)

```c
x == y
```

3 Pointer Arithmetic

3.1 Consider the following blocks of C code:

```c
void printall(int *x) {
    // Suppose x points to 0xABDE2464
    const int NUM_ELEMS = 3;
    for(int i = 0; i < NUM_ELEMS; i += 1) {
        printf("Address: %x \n", x);
        x++;
    }
}
```

(a) What three memory addresses are printed by this program?

0xABDE2464, 0xABDE2468, 0xABDE246C.

```c
void printall(char *x) {
    // Suppose x points to 0xABDE2464
    const int NUM_ELEMS = 3;
    for(int i = 0; i < NUM_ELEMS; i += 1) {
        printf("Address: %x \n", x);
        x++;
    }
}
```

(b) What three memory addresses are printed by this program?

0xABDE2464, 0xABDE2465, 0xABDE2466.

4 Programming with Pointers

4.1 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.

```c
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;
}
```

4.2 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.

```c
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.

```c
void increment(char* string) {
    for (i = 0; string[i] != 0; i++)
        string[i]++; // or (*(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) Copies the string `src` to `dst`.

```
void copy(char *src, char *dst) {
    while (*dst++ = *src++);
}
```

No errors.

(d) 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`.

```
void cs61c(char *src, size_t length) {
    char *srcptr, replaceptr;
    char replacement[16] = "61C is awesome!";
    srcptr = src;
    replaceptr = replacement;
    if (length >= 16) {
        for (int i = 0; i < 16; i++)
            *srcptr++ = *replaceptr++;
    }
}
```

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

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