CS 61C Fall 2022

C and Pointers

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 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!

2 Bit-wise Operations

2.1 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 $\wedge$ b**	**$\sim$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` $\wedge$ `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) $\sim$ ((`b | c`) `& a`)

`0b0111 1110`

3 Pass-by-who?

3.1 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.

```
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 4 lines long.

```
int mystrlen(char* str) {
    int count = 0;
    while (*str++) {
        count++;
    }
    return count;
}
```

4 Debugging

4.1 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.

```
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.

```
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`.

```
1  void copy(char *src, char *dst) {
2      while (*dst++ = *src++);
3  }
```

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`.

```
1  void cs61c(char *src, size_t length) {
2      char *srcptr, replaceptr;
3      char replacement[16] = "61C is awesome!";
4      srcptr = src;
5      replaceptr = replacement;
6      if (length >= 16) {
7          for (int i = 0; i < 16; i++)
8              *srcptr++ = *replaceptr++;
9      }
10 }
```

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

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