*Your Name (first last)*

# UC Berkeley CS61C
# Spring 2020 Midterm 1
## Solutions

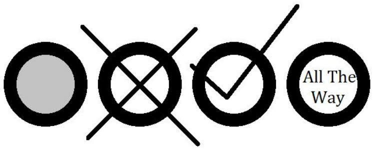
*SID*

*← SID & Test # To The Left (or aisle)*

*SID & Test # To The Right (or aisle) →*

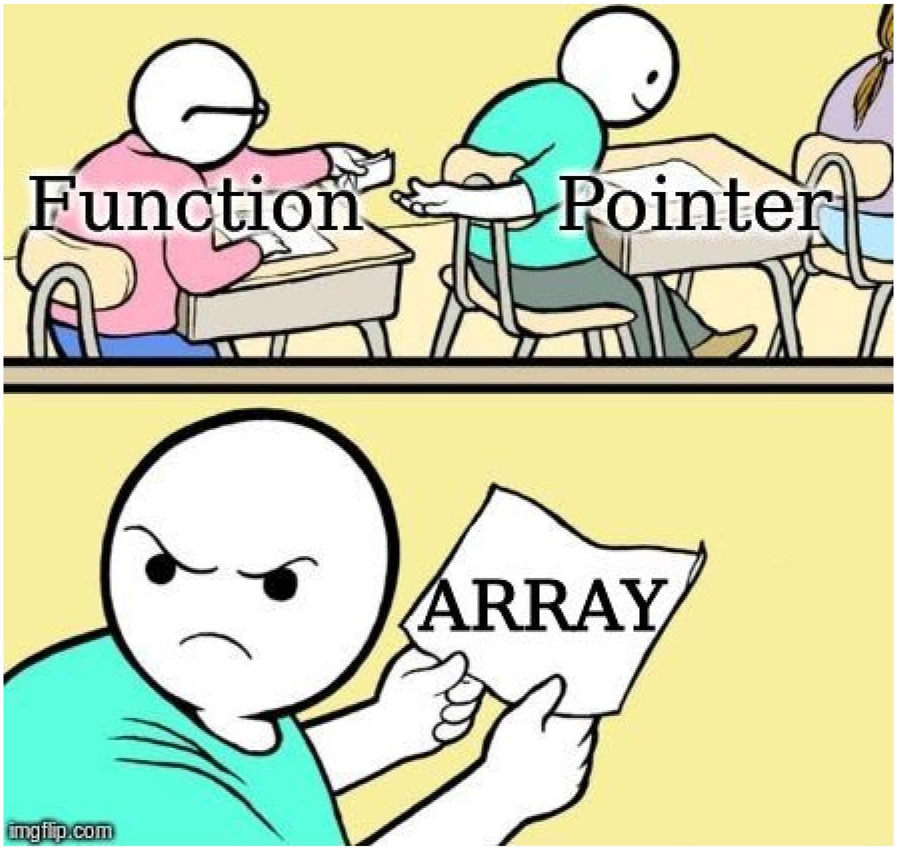
*TA name*

**Make sure to bubble in your answers all the way*…like this:***
**● *(select ONE)*, and ■*(select ALL that apply)*.**
**The following are examples of bubbles that *won't* be considered bubbled.**



*I am aware of the Berkeley Campus Code of Student Conduct and acknowledge that any academic misconduct will be reported to the Center for Student Conduct, and may result in partial or complete loss of credit and penalties that can include an F in the class. I am also aware that Nick Weaver takes cheating personally. I also promise to either give away my first-born child to the kelpies that live in Strawberry Creek or write [sic] after my signature to show that I actually do read the fine print...*

*Sign your [sic] name:* ________________________________________________



# Midterm 1 Clarifications

- 5b) or POSITIVE if the first element is bigger.  Also, the numbers themselves are all positive, <2^25, and separated by at least 2.

**<u>Q1) *String Cheese*</u> (8 pts = 2 * 1 + 3 + 3)**

Mark the correct lines that will allow the program to execute as specified below: There may be multiple correct answers.

a) Correctly gets the number of bytes in a string, including the null-terminator (Mark all that apply)

```
int get_strlen(char* str) {
        □ return strlen(str);
        ■ return strlen(str) + 1;
        □ return sizeof(str);
        □ return sizeof(str) + 1;
        □ return str.strlen() + 1;
        □ None of the above
}
```

Because strlen only returns the length of the string without the null-terminator, we must add 1

b) Gets the ith element of an array

```
int get_elem(int* arr) {
        ■ return arr[i];
        □ return arr + i;
        ■ return *(arr + i);
        □ return arr.get(i);
        □ return *arr + i;
        □ None of the above
}
```

Either use index notation or move the pointer and dereference

The following code is executed on a 32-bit little-endian system.

```
#include <stdio.h>
int main() {
    int doThis = 0x6C697665;
    char *dont = (char *)(&doThis);
    printf("A: ");
    for (int i = 0; i < 4; i++) {
        printf("%c", dont[i]);
    }
    printf("\n");
}
```

c) What is printed when this program is run? If it crashes/segfaults, write **n/a**.

**A: evil**

Doesn't segfault b/c C treats data as bits to be interpreted with regards to their type and trusts that the programmer's type casts are correct. Little-endian means least significant byte (0x65) is at the lowest memory address, so the answer is evil instead of live.

Carefully read the following code.

```
0  #include <stdio.h>
1  #include <string.h>
2  int main() {
3      char *boo = “go cardinals!”;
4      char *cheer = “go bears!!!!”;
5      printf(“%s”, cheer);
6      for (int i = 0; i < strlen(cheer); i++) {
7          boo[i] = cheer[i];
8      }
9      printf(“%s”, boo);
10 }
```

d) Does the program crash? If the program does not crash, write exactly what is printed to stdout. If the program crashes, identify both the line # that crashes and the line # you would fix to solve the crash

| | | |
|---|---|---|
| ● Yes | Line # where the crash occurs<br>○ 0 ○ 4 ○ 8<br>○ 1 ○ 5 ○ 9<br>○ 2 ○ 6 ○ 10<br>○ 3 ● 7 | Line # you would change to solve the crash.<br>○ 0 ○ 4 ○ 8<br>○ 1 ○ 5 ○ 9<br>○ 2 ○ 6 ○ 10<br>● 3 ○ 7 |
| ○ No | ____________________ | |

Crashes because string literals (“abcd”) are READ ONLY and cannot be modified. Changing line 3 to be char boo[] = “go cardinals!”; would fix it as the string literal would then be used to initialize a char array on the stack which is modifiable.

**Q2) *Number RIP* (8 pts = 8 * 1)**

Please fill out the following table. Write **N/A** if the conversion is not possible. Some entries have already been filled out for you. You may assume all binary numbers are 8 bits.

| Decimal | Binary (Two's complement) | Octal (base 8, two's complement) | Hex (two's complement) | Binary (Biased w/ added bias of -127) |
|---|---|---|---|---|
| -29 | 0b11100011 | 343 | 0xE3 | 0b01100010 |
| 89 | 0b01011001 | 131 | 0x59 | 0b11011000 |

| Decimal | Binary (Two's complement) | Octal (base 8, two's complement) | Hex (two's complement) | Binary (Biased w/ added bias of -127) |
|---|---|---|---|---|
| -29 | We first find the binary representation of 29, which is 0b00011101. We then flip all the bits and add 1, and that gives us 0b11100011 | Notice that 8 = $2^3$. Thus, we can group 3 binary digits at a time and represent them as one octal digit. Since 8 is not a multiple of 3, we can add a zero at the beginning of our binary number as this will not change the value. Now 0b 011 100 011 = $343_8$. | We group the binary digits in groups of 4. 0b 1110 0011 = 0xE3 | To go from a decimal value to its biased notation, we first subtract the bias from -29, and then represent the resulting number -29 - (-127) = 98 in its binary representation 0b01100010. |
| $1 * 8^0 + 3 * 8^1 + 1 * 8^2 = 89$ | We will take each octal digit and replace it with 3 binary digits, this gives us $131_8$ = 0b 001 011 001. Then we take the least significant 8 bits and get 0b01011001 | 131 | We group the binary digits in groups of 4. 0b 0101 1001 = 0x59 | We first subtract the bias -127 from 89 and get 216. 216 represented as an unsigned binary number is 0b11011000 |

**<u>Q3) *Unions*</u> (8 pts = 4 * 2)**

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

union Fun {
        uint8_t u[4];
        int8_t i[4];
        char s[4];
        int t;
};

int main() {
        union Fun *fun =
          calloc(1, sizeof(union
Fun));

        fun->i[0] = -1;

        //Question a
        printf("%u\n", fun->u[0]);

        fun->u[0] *= 2;

        //Question b
        printf("%d\n", fun->i[0]);

        fun->t *= -1;
        fun->t >>= 1;

        //Question c
        printf("%d\n", fun->i[1]);

        fun->s[0] = '\0';

        //Question d
        printf("%d\n", fun->t);

        free(fun);
        return 0;
}
```

Write what each print statement will print out in the corresponding box. Assume that this system is little-endian and that right shifts on signed integers are arithmetic.

a)

**255**

Because we are asking for the unsigned representation of the first byte in the union, we will get the value 255 instead of -1.

b)

**-2**

Multiplying u[0] by -2 multiplies i[0] by -2 as well (since they share the same bytes). Thus, the value printed is -2.

c)

**-1**

We negate all the bytes in the union and right shift it by one. This gives us 0xffffff81. Because of little-endianness, we want the second to last element– which is 0xff. This is equivalent to -1 in 8-bit signed decimal.

d)

**-256**

We set the firstmost byte in the union to "0". As the remaining bytes are all 1 still, this means that the remaining number is 0xffffff00. Converting this number to its positive equivalent in 32-bit two's complement will net us 0x100, which is equivalent to 256. Thus, this will print out -256.

**Q4) *CS61TREE Memory!*** **(8 pts = 7 * 0.5 + 3 + 1.5)**

For this problem, assume all pointers and integers are **four bytes** and all characters are **one byte**. Consider the following C code (all the necessary `#include` directives are omitted). C structs are properly aligned in memory and all calls to malloc succeed. For all of these questions, assume we are analyzing them right before main returns.

```
 typedef struct node {           node* newNode(void *data) {
     void *data;                   node *n = (node *) malloc(sizeof(node));
     struct node *left;            n->data = data;
     struct node *right;           n->left = NULL; n->right = NULL;
 } node;                           return n; }

int main() {
  char *r     = “CS 61C Rocks!”;
  char s[]    = “CS 61C Sucks!”; /* Reddit review... Warning: Nick sh*tposts too! */
  node nl;
  nl.data = (void *) r;
  node *root  = newNode((void *) &main);
  root->left  = malloc(strlen(r) + 1);
  root->right = newNode((void *) s);
  root->right->left  = newNode((void *) r);
  root->right->right = newNode((void *) &printf);
  root->left  = &nl;
}
```

a) Each of the following evaluate to an address in memory. In other words, they "point" somewhere. Where in memory do they **point**?

| | *Code* | *Static* | *Stack* | *Heap* |
|---|---|---|---|---|
| **`root`**<br>The root node is malloced in newNode so it will be stored in the heap. | ○ | ○ | ○ | ● |
| **`root->data`**<br>We passed in a pointer to the main function which is stored in the code. | ● | ○ | ○ | ○ |
| **`root->left`**<br>At the end of main, we set the `root->left` node to the address of `nl` which was created on the stack. | ○ | ○ | ● | ○ |
| **`root->left->data`**<br>We set the data in the nl data structure to be r. Since `r` was declared a `char *`, it is a pointer to a static string thus it will be in | ○ | ● | ○ | ○ |

b) How many bytes of memory are allocated but not **`free()`**d by this program, if any?

**`62 Bytes`**

*We malloc a total of 4 nodes. Each node is 12 bytes in size since we have 3 pointers and all pointers are 4 bytes.*

*We also malloced some data to root->left of size strlen(r) + 1 = 13 + 1 = 14.*

*Since we do not free any of those pointers, we will leak 4 * 12 + 14 bytes of data = 48 + 14 = 62 Bytes.*

| | | | | | |
|---|---|---|---|---|---|
| static memory. | | | | | |
| **root->right->data**<br>root->right was created with setting the data to s. If you look at the way s was declared (char [ ]), this means it was placed on the stack. Thus it is pointing to the stack. | ○ | ○ | ● | ○ | |
| **root->right->left->data**<br>This node has the same reasoning as root->left->data | ○ | ● | ○ | ○ | |
| **&newNode**<br>newNode is located in the code since it is a function which will execute. | ● | ○ | ○ | ○ | |

```
void free_tree(node *n) {
  if (n == NULL) return;
  free_tree(n->left);
  free_tree(n->right);
  free(n);
}
```

c) Given this free function, if we called `free_tree(root) after all the code in main is executed`, this program would have well defined behavior.

○ True
● False

This `free_tree` function would operate correctly SO LONG as every node was allocated correctly (with `malloc` or `calloc`). Since we see that we allocated `root->left` on the heap, if we called `free_tree(root)`, we will end up freeing an address on the stack which is undefined behavior!

**Q5) *The Bananananananana Hunt*** **(15 pts = 4 + 4 + 7)**

You’re on a hunt around campus to find the best fresh banana available. You find a note from the CS61C course staff with clues, but they’re encrypted so that only the best students can find the bananas. **Note that the solutions for each part are not dependent on the other parts.**

(a) Your first clue is a string encoded in an integer array `info` of length `len`. We encoded the null-terminated string by placing the *i*th character in the most significant byte of *i*th integer in `info`. Modify the code below so that the original string is properly printed and so that there are **no memory leaks or undefined behavior**.

For this question, `len` is the size of all the characters needed for a properly formatted string (all the letters and the null terminator). The first step is to allocate the memory for this buffer on the heap, using `malloc` or `calloc`. Next, since the information is encoded in the most significant bit, a right shift will move the character into the least significant bit so that a cast to `char` type will keep the data. Since `ints` are 4 bytes, the left shift must be by three bytes (24 bits). Finally, the buffer created to print the information must be freed to avoid a memory leak.

```
void clue1(unsigned int* info, int len) {
    char* info_to_print = malloc (len * sizeof(char));
    for (int i = 0; i < len; i++) {
        info_to_print[i] = (char) info[i]>>24 /* Others possible as well */;
    }
    printf(“%s\n”, info_to_print);
    free(info_to_print);
}
```

(b) Having discovered the identity, you follow it and find a large array of double precision floating point (type **double**). The clue says you want the 5th smallest element casted to an integer. True, you could just go through the array but, being a proper CS student, you decide to first sort the array using a library function and then take the 5th element. Fortunately, C has a quicksort function in the standard library:

```
void qsort ( void * base, size_t num, size_t size,
             int ( * comparator ) ( const void *, const void * ) );
```

That is, the function takes four arguments: a pointer to the array, the total number of elements, the size of each element of the array, and a comparison function. The comparison function should return negative if the first element is less than the second, 0 if they are the same, or positive if the first element is bigger. Your code should compile without warnings.

```
int comp(void *p1, void *p2){
    double a = *((double *) p1);
    double b = *((double *) p2);
    return a-b ; /* C will cast a double to an int automagically */
}
void clue2(double* info2, int len) {
    qsort(info2, len, sizeof(double), &comp);
    printf("%i\n", (int) info2[4] );
}
```

(c) You arrive at the room, only to find a door locked with a keycode. Spray painted on the wall, you see

*“How many stairwells have a power-of-two number of steps? Print the answer* ***in hex****...”*

So close to your goal, you crowdsource this question to your favorite social media. Enlisting a friend taking CS 186, you end up with an array of step counts for all stairs which are all positive integers. Create a function to see the total number of stairwells with exactly a power of 2. Hint: you know ***X*** is a power of 2 if and only if ***X*** and ***X-1*** have no bits in common and ***X*** is nonzero. You do not need to use all the lines.

There are multiple valid methods to approach this question. The staff solution requires the least number of lines, and uses the hint that x and x-1 have no bits in common for a power of 2. A power of 2 is found for any non-zero value where the logical and of x and x-1 results in a zero value (thus `stairs[i]` is checked to ensure a non-zero value, then `(stairs[i] & (stairs[i]-1))` is tested for a zero value. Another way to check if the entry is a power of two which is possible with the given lines is to test each bit within the entry and make sure only one bit has a 1 value. Finally, `printf` is called with %x, printing the number of entries in hexadecimal (given by the chart below).

```
void pows_of_2(unsigned int* stairs, int len) {
    int matching_entries = 0;
    __________________________________________________;
    __________________________________________________;
    for (int i = 0; i < len; i++) {
        ______________________________________________;
        if (stairs[i] && (stairs[i] & (stairs[i]-1)) == 0) {
            matching_entries += 1;
        }
        ______________________________________________;
    }
    printf(“%x\n”, matching_entries );
}
```

Print format specifier table.

| **Specifier** | **Output** | **Specifier** | **Output** |
|---|---|---|---|
| **d or i** | Signed decimal integer | **E** | Scientific notation (mantissa/exponent), uppercase |
| **u** | Unsigned decimal integer | **g** | Use the shortest representation: %e or %f |
| **o** | Unsigned octal | **G** | Use the shortest representation: %E or %F |
| **x** | Unsigned hexadecimal integer | **a** | Hexadecimal floating point, lowercase |
| **X** | Unsigned hexadecimal integer (uppercase) | **A** | Hexadecimal floating point, uppercase |
| **f** | Decimal floating point, lowercase | **c** | Character |
| **F** | Decimal floating point, uppercase | **s** | String of characters |
| **e** | Scientific notation (mantissa/exponent), lowercase | **p** | Pointer address |

**Q6) *Fl_at___ P_int_u_bers* (9 pts = 3 * 3)**

You received a sequence of IEEE standard 16-bit floating point numbers from your friend. So you don't need to look it up on your green sheet, we will remind you that a 16-bit floating point is **1 sign bit, 5 exponent bits, and 10 mantissa bits**. The bias for the exponent is **-15**.

Unfortunately, cosmic rays corrupted some of the data, rendering it unreadable. For the following problems, we will use "x" to refer to a bit that was corrupted (in other words, we don't know what the sender wanted that bit to be). For example, if I received the data "0b0xx1", the sender sent one of "0b0001", "0b0101", "0b0011", or "0b0111".

a) You receive the data "0b0x1x0x1x0x1x0x1x". What is the **hexadecimal encoding** of the **biggest number** the sender could have sent?

0x7777

Answer: 0x7777. One property of floating point numbers is that their order is the same as that of sign-magnitude integers (ignoring NaNs); for example, 0x5555 < 0x7555. In order to maximize the number, we therefore want to set all the "x"s to 1. This yields the encoding 0x7777.

b) You receive the data "0b1110xxxxxxxxxxxx". What is the **decimal value** of the **smallest number** the sender could have sent (i.e. it is less than all of the other possibilities)? You must provide the decimal form, ***do not leave as a power of 2***.

-8188

Answer: -8188. By the previous observation, the smallest number is encoded by 0xEFFF. This has sign bit 1, exponent 0b11011 - 15 = 12, and mantissa (1).1111111111 = 2-2^-10. Our answer is thus -4096*(2-2^-10) = -8192+4=-8188

c) For the next number, the sign and exponent are correct but all of the mantissa was corrupted. The sender did not send a NaN or infinity. What is the **smallest possible** positive number the sender could have sent **as a power of 2**?

$2^{-24}$

The smallest possible power of 2 is when we receive the bits "0b000000xxxxxxxxxx", with the corrupted bits being filled by "0b0000000000000001". This is equal to 2^-14 * 2^-10 = 2^-24.

**Q7) *RRIISSCC-VV*** **(12 pts)**

In this question, you will implement a simple recursive function in RISC-V. The function takes a decimal number as input, then outputs it's binary representation encoded in the decimal digits.

```
int findBinary(unsigned int decimal) {
      if (decimal == 0) {
            return 0;
      } else {
            return decimal % 2 + 10 * findBinary(decimal / 2);
      }
}
```

For example, if the input to this function is 10, then the output would be 1010.

```
findBinary:
    addi sp, sp, -8            # preamble... a0 will have arg and be where we return
    sw ra, 4(sp)               saving return addresses on the stack
    sw s0, 0(sp)               saving "decimal % 2" on the stack
    beq a0, x0, postamble      # base case, we will just return 0
    andi s0, a0, 1             # set s0 to a0 % 2, "decimal % 2"
    srli a0, a0, 1             # set a0 to a0 / 2, "decimal / 2"
                                 new argument for recursive call
    jal ra, findBinary         # recursive call
    li t0, 10/addi t0, x0, 10  # Load the value 10 into t0
    mul a0, t0, a0             # a0 = a0 * 10,
                                 a0 = 10 * accumulated return values from recursive calls
    add a0, a0, s0             # a0 = s0 + a0,
                                 accumulating the stored return value "decimal % 2"
                                 With return value from recursive calls
postamble:
    lw ra, 4(sp)  # Restore ra  loading back previously saved return addresses
    lw s0, 0(sp)  # restore s0  loading back previously saved "decimal % 2" from stack
    addi sp, sp, 8   # restore sp
end:
    jr ra
```

The recursive part of the function stores all of the first part of the return value "decimal % 2" on the stack
The second part of the function and postamble are combing all the return values by "+" and "10 * "

Good Luck, and ***Don't F@#)(* It Up!***