Weaver | CS 61C | Midterm 1
Spring 2020 | Machine Structures

PRINT your name: ____________________, ____________________
(first) (last)

PRINT your student ID: ______________________________

Read the following honor code and sign your name.

I understand that I may not collaborate with anyone else on this exam, or cheat in any way. I am aware of the Berkeley Campus Code of Student Conduct and acknowledge that academic misconduct will be reported to the Center for Student Conduct and may further result in, at minimum, negative points on the exam and a corresponding notch on Nick's Stanley Fubar demolition tool.

SIGN your name: ______________________________

You have 110? minutes. There are 7 questions of varying credit, for a total of 68 points.
For questions with **circular bubbles**, you may select only one choice.

○ Unselected option

● Only one selected option

For questions with **square checkboxes**, you may select one or more choices.

■ You can select

■ multiple squares

Anything you write that you ~~cross out~~ will not be graded. Anything you write outside the answer boxes will not be graded.

Question 1 ***String Cheese*** **(8 points)**

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

Q1.1 (1 point) 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

Solution:
Because `strlen` only returns the length of the string without the null-terminator, we must add 1. `sizeof` won't work in this case because it would return the number of bytes that the `char*` takes, which does not tell us how many of bytes are in a string. `.strlen()`, which would be an instance method, does not exist in C (since strings are just a pointer to its first character).

Q1.2 (1 point) 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.get(i);`
- ☐ `return *arr + i;`
- ☐ None of the above

Solution: We can index into an array using either the index notation (`[i]`) or by moving the pointer and dereferencing the pointer. Adding `i` to `arr` only moves the pointer and does not dereference it, therefore it would return a pointer, not an `int`. Arrays in C do not have a `.get` method (or any other methods). `*arr + i` would dereference the array first, retrieving the first element, then add `i` to the first element (see C order of operations).

Q1.3 (3 points) 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");
}
```

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

Solution: The program prints out `A: evil`.
The program doesn't segfault because C treats data as bits to be interpreted with regards to their type and trusts that the programmer's type casts are correct. This means that we can treat an `int` as a series of `char`s. Little-endian means the least significant byte (0x65) is located at the lowest memory address, which is accessed first, so the program prints `evil`, not `live`.

Q1.4 (3 points) 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  }
```

[✓ Yes]	Line # where the crash occurs: 0 1 2 3 4 5 6 [✓ 7] 8 9 10	Line # to change to solve the crash: 0 1 2 [✓ 3] 4 5 6 7 8 9 10
No	Write exactly what is printed to stdout	

Solution: The program crashes because string literals (`"abcd"`) are **read only** and cannot be modified. On line 7, the program attempts to modify `boo`, which would cause the program to crash. To prevent the crash, line 3 can be changed to `char boo[] = "go cardinals!";`, which would use the string literal (`"go cardinals!"`) to initialize a `char` array, which is modifiable because it is located in the stack.

Editor's note: When reformatting this exam, we've intentionally left the line numbering for this question to be 0-indexed since it was also 0-indexed in the original version of this exam. The remainder of this exam will have line numbers that are 1-indexed, and are not referenced in questions (only referenced in the solutions).

Question 2 ***Number RIP*** **(8 points)**

Please complete each of the following parts. Write N/A if the conversion is not possible. You may assume all binary numbers are 8 bits.

Convert -29 (a decimal number) to the following representations:

Q2.1 (1 point) Binary (two's complement)

Solution: `0b11100011`

We first find the binary representation of 29, which is 0b00011101. We then flip all the bits and add 1, giving us 0b11100011.

Q2.2 (1 point) Octal (base 8, two's complement)

Solution: `343`

Since $8 = 2^3$, we can group 3 binary digits at a time and represent them as one octal digit. First, we can convert 29 to binary, giving us 0b11100011. Then, since this binary number only has 8 digits, we can pad it with a 0 so we can split into groups of 3, which gives us 0b 011 100 011. Then, convert each individual group into an octal digit, which gives us our answer of 343_8.

Q2.3 (1 point) Hex (two's complement)

Solution: 0xE3

Since $16 = 2^4$, we can group the binary digits in groups of 4, giving us 0xE3.

Q2.4 (1 point) Binary (biased with added bias of -127)

Solution: 0b01100010

Since `biased + bias = actual`, we can subtract the bias from -29, our number, to determine the number in this biased notation. Subtracting -127 from -29 gives us 98, and converting to binary gives us 0b01100010.

Convert 131_8 to the following representations:

Q2.5 (1 point) Decimal

Solution: 89

$1 \times 8^2 + 3 \times 8^1 + 1 \times 8^0 = 89$.

Q2.6 (1 point) Binary (two's complement)

Solution: 0b01011001
Since $8 = 2^3$, we can replace each octal digit with 3 binary digits, which gives us 0b 001 011 001. Since we're working with 8 bit binary numbers, we can ignore the most significant bit (which is 0 anyways), and arrive at 0b01011001.

Q2.7 (1 point) Hex (two's complement)

Solution: 0x59
Since $16 = 2^4$, we can group binary digits in groups of 4 and replace each group with the corresponding hexadecimal character. Grouping in 4s gives us 0b 0101 1001, which translates to 0x59.

Q2.8 (1 point) Binary (biased with added bias of -127)

Solution: 0b11011000
Starting at the decimal representation of 89, we subtract the bias, giving us 216. Then, we can express 216 as an unsigned binary number, which is 0b11011000.

Question 3 *Unions* **(8 points)**

```
#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;

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

    fun->u[0] *= 2;

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

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

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

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

    // Part (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.

Q3.1 (2 points)

Solution: 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.

Q3.2 (2 points)

Solution: -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.

Q3.3 (2 points)

Solution: -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.

Q3.4 (2 points)

Solution: -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.

Question 4 ***CS61TREE Memory!*** **(8 points)**

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.

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

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

Q4.1 (0.5 point) `root`

- ○ Code
- ○ Static
- ○ Stack
- [✓ **Heap**]

Solution: The `root` node is malloced in `newNode` so it will be stored in the heap.

Q4.2 (0.5 point) `root->data`

[✓ **Code**] Stack
Static Heap

Solution: We passed in a pointer to the `main` function which is stored in the code.

Q4.3 (0.5 point) `root->left`

Code [✓ **Stack**]
Static Heap

Solution: At the end of main, we set the `root->left` node to the address of `nl` which was created on the stack.

Q4.4 (0.5 point) `root->left->data`

Code Stack
[✓ **Static**] Heap

Solution: 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 static memory.

Q4.5 (0.5 point) `root->right->data`

Code [✓ **Stack**]
Static Heap

Solution: `root->right` was created with setting the `data` to `root->right`. 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.

Q4.6 (0.5 point) `root->right->left->data`

Code Stack
[✓ **Static**] Heap

Solution: This node has the same reasoning as `root->left->data`.

Q4.7 (0.5 point) `&newNode`

[✓ **Code**] Stack
Static Heap

Solution: `newNode` is located in the code since it is a function which will execute.

Q4.8 (3 points) How many bytes of memory are allocated but not `free()`d by this program, if any?

Solution: 62

We `malloc` a total of 4 nodes (lines 18, 20, 21, 22). Each node is 12 bytes in size since we have 3 pointers in each node and all pointers are 4 bytes. We also malloced some data to `root->left` of size `strlen(r) + 1` = 13 + 1 = 14 (line 19). Since we do not free any of those pointers, we will leak $4 * 12 + 14 = 62$ bytes of data. Note that data on the stack is automatically `free()`d, so the `node` allocated on like 16 does not count.

Q4.9 (1.5 points)

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

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]

Solution: 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 stack (it points to `nl`, allocated on line 16), if we called `free_tree(root)`, we will end up freeing an address on the stack which is undefined behavior!

Question 5 ***The Banananananananana Hunt*** **(15 points)**

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.**

Q5.1 (4 points) Your first clue is a string encoded in an integer array info of length `len`. We encoded the null-terminated string by placing the ith character in the most significant byte of ith 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**.

```
void clue1(unsigned int* info, int len) {

  char* info_to_print = ________________(_____________ sizeof(char));
  for (int i = 0; i < len; i++) {

      info_to_print[i] = (char) ____________________________________;
  }
  printf(\"%s\n", info_to_print);

  ____________________________________;
}
```

Solution:

Blank 1: `malloc`

Blank 2: `len *`

Blank 3: `info[i] >> 24` (others possible as well)

Blank 4: `free(info_to_print);`

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 `int`s 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.

Q5.2 (4 points) 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.

Clarification during exam: The numbers themselves are all positive, $< 2^{25}$, and separated by at least 2.

```
int comp(void *p1, void *p2){

    double a = _________________;

    double b = _________________;
    /* C will cast a double to an int automagically */

    return _____________________;
}
void clue2(double* info2, int len) {

    qsort(_____________________, __________________________,

    _____________________, __________________________);

    printf(\%i\n", ____________________________);
}
```

Solution:
Blank 1: `*((double *) p1)`
Blank 2: `*((double *) p2)`
Blank 3: `a - b` Blank 4: `info2`

Blank 5: `len`
Blank 6: `sizeof(double)`
Blank 7: `&comp`
Blank 8: `(int) info2[4]`

Q5.3 (7 points) 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.

```
void pows_of_2(unsigned int* stairs, int len) {
  int matching_entries = 0;

  ________________________________________________;

  ________________________________________________;
  for (int i = 0; i < len; i++) {

    ____________________________________________;

    if (______________________________________) {
        matching_entries += 1;
    }

    ____________________________________________;
  }

  printf(_______________, ____________________);s
}
```

Solution:

Blank 1: empty
Blank 2: empty
Blank 3: empty
Blank 4: `stairs[i] && (stairs[i] & (stairs[i]-1)) == 0`
Blank 5: empty
Blank 6: `"%x\n"`
Blank 7: `matching_entries`

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).

Print Format Specifier Table

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

Question 6 ***Fl at P int u bers*** **(9 points)**

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".

Q6.1 (3 points) You receive the data "0b0x1x0x1x0x1x0x1x". What is the **hexadecimal encoding** of the **biggest number** the sender could have sent?

> **Solution:** 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.

Q6.2 (3 points) 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***.

> **Solution:** -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 $-4096 * (2 - 2^{-10}) = -8192 + 4 = -8188$.

Q6.3 (3 points) 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**?

> **Solution:** 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 $2^{-14} * 2^{-10} = 2^{-24}$.

Question 7 ***RRIISSCC-VV*** **(12 points)**

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)
    sw s0, 0(sp)

    beq a0, x0, _____________  # base case, we will just return 0

    _________________________  # set s0 to ??? with a bitwise op

    _________________________  # set a0 to ??? with a bitwise op

    jal ra, _________________  # recursive call

    _________________________  # load the value 10 into t0
    mul a0, t0, a0             # a0 = a0 * 10

    _________________________  # a0 = ???
postamble:

    _________________________  # restore ra

    _________________________  # restore s0

    _________________________  # restore sp
end:
    jr ra

```

Solution:

Blank 1: `postamble` (base case, returns 0)
Blank 2: `andi s0, a0, 1` (stores `decimal % 2` into `s0`)
Blank 3: `srli a0, a0, 1` (stores `decimal / 2` into `a0`)
Blank 4: `findBinary` (recursive call)
Blank 5: `li t0, 10` or `addi t0, x0, 10` (sets `t0` equal to 10)
Blank 6: `add a0, a0, s0` (calculates `decimal % 2 + 10 * findBinary(decimal / 2)`)
Blank 7: `lw ra, 4(sp)` (restore `ra`)
Blank 8: `lw s0, 0(sp)` (restore `s0`, which is the previously saved ”`decimal % 2`”)
Blank 9: `addi sp, sp, 8` (restore `sp`)

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 combining all the return values by ”+” and ”10 * ”.