CS107 Cynthia Lee
Autumn 2018 November 2, 2018

CS107 Midterm Examination

You have **1 hour 50 minutes** to complete all problems.

This is a closed book, closed computer exam. You are allowed only 1 page (8.5x11, both sides) of notes, and no other aids. You don't need to #include any libraries, and you needn't use assert to guard against any errors. Understand that the majority of points are awarded for concepts taught in CS107, and not prior classes. You don't get many points for for-loop syntax, but you certainly get points for proper use of &, *, and the low-level C functions introduced in the course. The last page of the exam is a reference sheet. DO NOT ADD OR REMOVE PAGES OF THE EXAM (exception: you may remove the reference sheet). If you need extra space for scratch and/or problem responses, use the blank back sides of each page. Changing the pages confuses the online grading system and will cause you to lose points.

I appreciate your extraordinary effort this quarter and trust it will pay off here. You've got this!

IMPORTANT: WRITE YOUR NAME AND SUNET (that is your myth login) ON EVERY PAGE

Please do this at the beginning of the exam time to ensure you won't be scrambling at the end. Late exam submissions ("I just need to write my name on each page!") will **NOT** be accepted. Thanks for your cooperation.

NAME: _____ SUNET: _____

I accept the letter and spirit of the honor code. I will neither give nor receive unauthorized aid on this exam.

[signed] _____

Problem 1: Integer Representation, 7pts
Problem 2: Pointers and Arrays, 8pts
Problem 3: Memory Diagram, 13pts
Problem 4: Generics, 13pts
Problem 5: Bitwise Operations, 8pts
Write your name and SUNet on each page, 1pt
TOTAL: 50 points

LAST (FAMILY) NAME: _____ SUNET: _____

Problem 1: Integer Representation (7pts)

There is a small amount of scratch space between problems for you to write your work, but it is not necessary to show your work, nor will it be evaluated. (You may also use the blank back sides of each exam page as scratch space.) Please write your answer in the provided box only.

(a) (2pts) Consider the code `char c = 'm';` Write the binary representation of the value of `c`, with exactly the right number of bits for the type (i.e., including leading zeros, if any). There is an ASCII table at the end of the exam.

(b) (2pts) Our state's postal abbreviation is CA. If we interpret `0xCA` as a *signed* 8-bit value (i.e, type `char`), what is the value in decimal?

(c) (2pts) Write the decimal number -31 in 8-bit signed (two's complement):

Halloween-related fun fact:

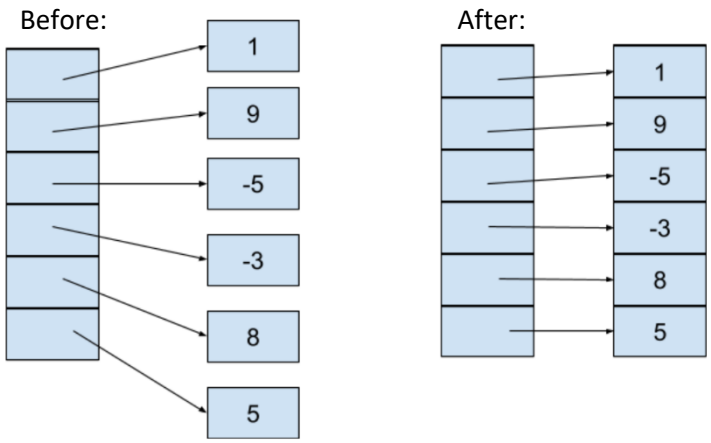
Each year, the dates of Christmas and Halloween are Dec-25 and Oct-31, respectively. The way to write **Dec**imal **25** in **Oct**al (base 8) is…wait for it...**31**! You're welcome.

(d) (1pt) What is your favorite hexadecimal number?

LAST (FAMILY) NAME: _____ SUNET: _____

Problem 2: Pointers and Arrays (8pts)

Write a function `clean_up_ptrarr`. It takes an array of pointers, each pointer points to one heap-allocated integer. Each integer's memory was acquired by a separate `malloc` call (no two the same), so they could be spread all over the heap, not in one contiguous block of memory like an array. Your function should "clean up" this situation, by copying them all into one contiguous array, updating the input array's pointers so they point to the same integer value but in its newly cleaned-up contiguous array (see "before & after" diagram, below).



Specifically, your function should:

(1) create a new heap-allocated array that holds type `int`;

(2) copy the integer values pointed-to by the pointers in the input array to the newly-allocated `int` array, preserving the same order;

(3) then set the input array's pointers to point to the values in their new locations.

Notes: Since we aren't using the `int` values that were pointed-to by the original input array anymore, you should make sure that **free the memory** associated with them at the appropriate time. Only write in the boxes. **You may not need every box.**

```
void clean_up_ptrarr (int *arr[], size_t nelems) {

    [                                                    ]

    int *copy = malloc( [                                ] );
    for (int i = 0; i < nelems; i++) {
        [                                                ]
        [                                                ]
        [                                                ]
    }
    [                                                    ]
}
```

LAST (FAMILY) NAME: _____ SUNET: _____

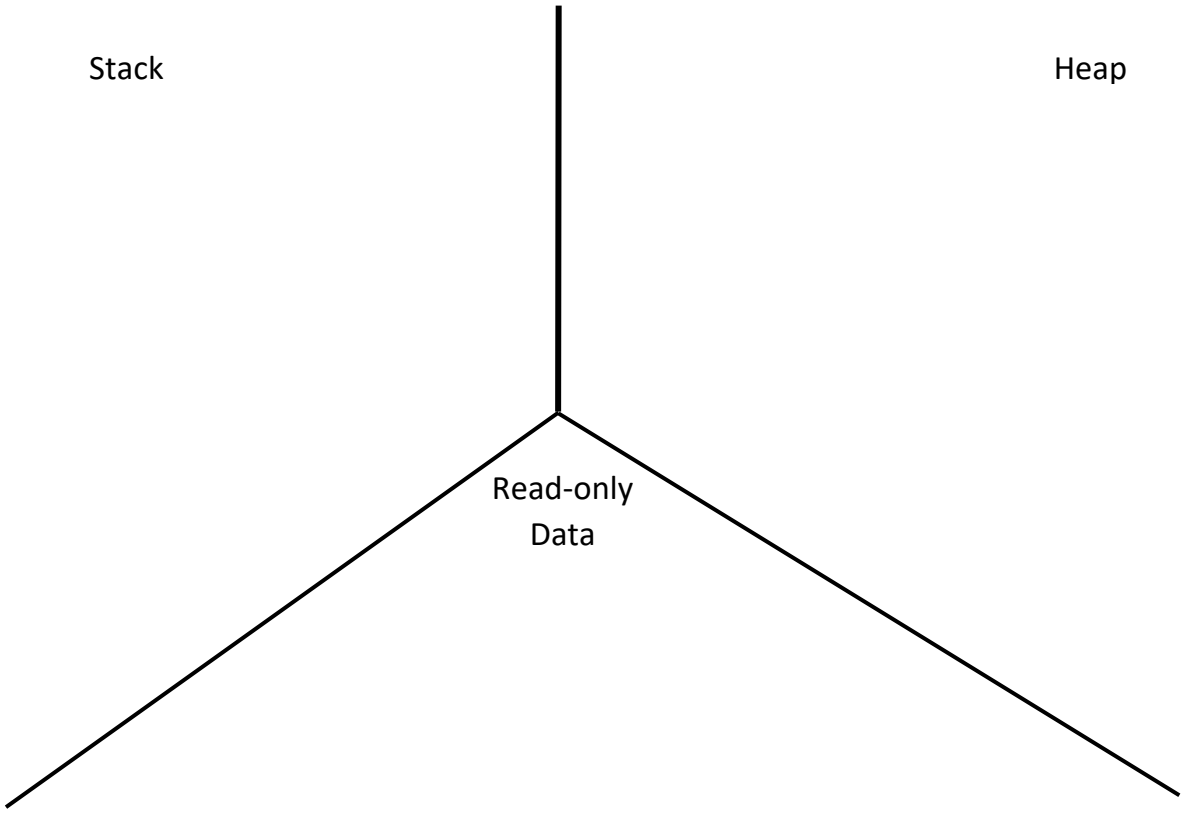
Problem 3: Memory Diagram (13pts)

For this problem, you will draw a memory diagram of the state of memory (like those shown in lecture) as it would exist at the end of the execution of this code:

```
char *aaron = "burr, sir";
int *the_other = malloc(12);
the_other[0] = 51;
char *eliza[2];
*eliza = strdup("satisfied");
*(int *)((char *)the_other + 4) = 85;
aaron++;
eliza[1] = aaron + 2;
```

Instructions:

- Place each item in the appropriate segment of memory (stack, heap, read-only data).
- Please write array index labels (0, 1, 2, ...) next to each box of an array, in addition to any applicable variable name label. (With the array index labels, it doesn't matter if you draw your array with increasing index going up or down--or sideways for that matter.)
- Draw strings as arrays (series of boxes), with individual box values filled in appropriately and array index labels as described above.
- Take care to have pointers clearly pointing to the correct part of an array.
- Leave boxes of uninitialized memory blank.
- NULL pointer is drawn as a slash through the box, and null character is drawn as `'\0'`.



LAST (FAMILY) NAME: _____ SUNET: _____

Problem 4: Generics and Function Pointers (13pts)

For this problem, you will write a generic function that takes an array and selectively removes items from it according to a client-specified criteria. The behavior of this function is as follows:

- Signature: `void remove_less(void *arr, size_t *nelems, size_t width, cmp_fun_t cmp)`. `cmp_fun_t` is defined the same way as on assign4 and in part (b) below.
- `remove_less` removes all elements that are less than the first element in the array. If the array has size 0 or 1, then there are no elements to remove, so do nothing.
- The parameter `nelems` is passed by reference and should be updated if necessary to reflect the new size after any removals done by `remove_less`.
- When the function is done, the array will still have the same capacity (do not call `realloc`). The order of elements that remain should be preserved.
- Just like on assign4, **for full credit your solution should use appropriate C library functions** rather than re-implement that functionality (for example, with unnecessary loops). Remember that a C library reference is at the end of the exam.
- **Example:** If given input array `{5, 1, 2, 7, 5, 3, 8, 0}` and the usual `int` comparison function (see part (b) for a reference solution), then, at return time, the array should be `{5, 7, 5, 8}` and the number of elements should be updated to 4.

(a) (10pts) Write the `remove_less` function. We have written some of it for you, and you must follow the structure we started, even if you might think of other ways of doing it.

```
void remove_less (void *arr, size_t *nelems, size_t width, cmp_fun_t cmp) {

   for (size_t i = [          ]; i > 0; i--) { // note i--
      // First calculate the address of the ith array element
      void *ith = [                                    ];

      // Compare the ith and 0th items
      int res = cmp([                                    ]);

      // If ith < 0th, remove ith by moving elements over. Do not move more
      // elements than necessary. Complete the function here.
      if (res < 0) {
```

[ ]

LAST (FAMILY) NAME: _____ SUNET: _____

(b) (3pts) On assign4, you had practice writing your own `cmp_fun_t` comparison functions, which are defined as follows:

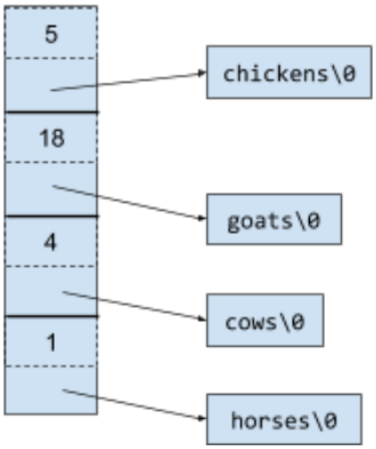
```
typedef int (*cmp_fn_t)(const void *p, const void *q);
```

As a reminder of how they work, here is a reference implementation of a comparison function for elements of type `int`:

```
int int_compare(const void *p, const void *q) {
    return *(const int *)p - *(const int *)q;
}
```

Now consider the following definition of a struct used for tracking populations of various types of pets on my fantasy farm:

```
struct farm { size_t count; char *species; };
```



To the right is a memory diagram of an array of these structs (array size 4). Write a comparison function for this struct that takes the sum of the `count` and the length of the string `species`, and compares those values.

- **Example:** comparing the top two elements in the diagram at right would compute (5+8) and (18+5), determining that 13 < 23.

```
int farm_compare(const void *p, const void *q) {
```

```
}
```

LAST (FAMILY) NAME: _____ SUNET: _____

Problem 5: Bitwise Operations (8pts)

Write a function that takes an `unsigned int` and returns true if its binary representation contains at least one instance of **at least two consecutive zeros**. Examples:

- Input: `00110111101111101111111111011111` Return: `true`
- Input: `11111101111011111110000111111111` Return: `true`
- Input: `01010101010101010101010101010101` Return: `false`
- Input: `11111111111111111111111111111111` Return: `false`

(a) (6 pts) Write a function that uses a loop to each pair of bits to detect a pair of zeros.

```
bool zeros_detector(unsigned int n) {
```

```
}
```

(b) (2pts) Now write the same `zeros_detector` function but *without using any loops or recursion* (and no excessive writing the same lines of code to evade this). The small point value understates the difficulty of this question, so allocate your time wisely (the intention is to minimally penalize students who solve part (a) but not part (b)).

```
bool zeros_detector(unsigned int n) {
```

```
}
```

LAST (FAMILY) NAME: _____ SUNET: _____

ASCII Table Reference

TABLE 3
ASCII CHARACTER CODES (DECIMAL)

0	Ctrl-@	32	Space	64	@	96	`
1	Ctrl-A	33	!	65	A	97	a
2	Ctrl-B	34	"	66	B	98	b
3	Ctrl-C	35	#	67	C	99	c
4	Ctrl-D	36	$	68	D	100	d
5	Ctrl-E	37	%	69	E	101	e
6	Ctrl-F	38	&	70	F	102	f
7	Ctrl-G	39	'	71	G	103	g
8	Backspace	40	(	72	H	104	h
9	Tab	41	)	73	I	105	i
10	Ctrl-J	42	*	74	J	106	j
11	Ctrl-K	43	+	75	K	107	k
12	Ctrl-L	44	,	76	L	108	l
13	Return	45	-	77	M	109	m
14	Ctrl-N	46	.	78	N	110	n
15	Ctrl-O	47	/	79	O	111	o
16	Ctrl-P	48	0	80	P	112	p
17	Ctrl-Q	49	1	81	Q	113	q
18	Ctrl-R	50	2	82	R	114	r
19	Ctrl-S	51	3	83	S	115	s
20	Ctrl-T	52	4	84	T	116	t
21	Ctrl-U	53	5	85	U	117	u
22	Ctrl-V	54	6	86	V	118	v
23	Ctrl-W	55	7	87	W	119	w
24	Ctrl-X	56	8	88	X	120	x
25	Ctrl-Y	57	9	89	Y	121	y
26	Ctrl-Z	58	:	90	Z	122	z
27	Escape	59	;	91	[	123	{
28	Ctrl-\	60	<	92	\	124	|
29	Ctrl-]	61	=	93	]	125	}
30	Ctrl-^	62	>	94	^	126	~
31	Ctrl-_	63	?	95	_	127	Delete