CS107 Cynthia Lee
Autumn 2018

CS107 Midterm Practice Problems

Problem 1: Integer Representation
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 on the provided lines.

a) Write the signed (two's complement) binary number 11101100 in decimal:
____

b) Write the unsigned binary number 1100011011 in hexadecimal:
____

c) Write the hexadecimal number 0x1DEAD as an unsigned binary number:

____

d) Write the decimal number 21 as an 8-bit binary number:
____

e) Write the decimal number -15 as an 8-bit signed (two's complement) binary number:

____

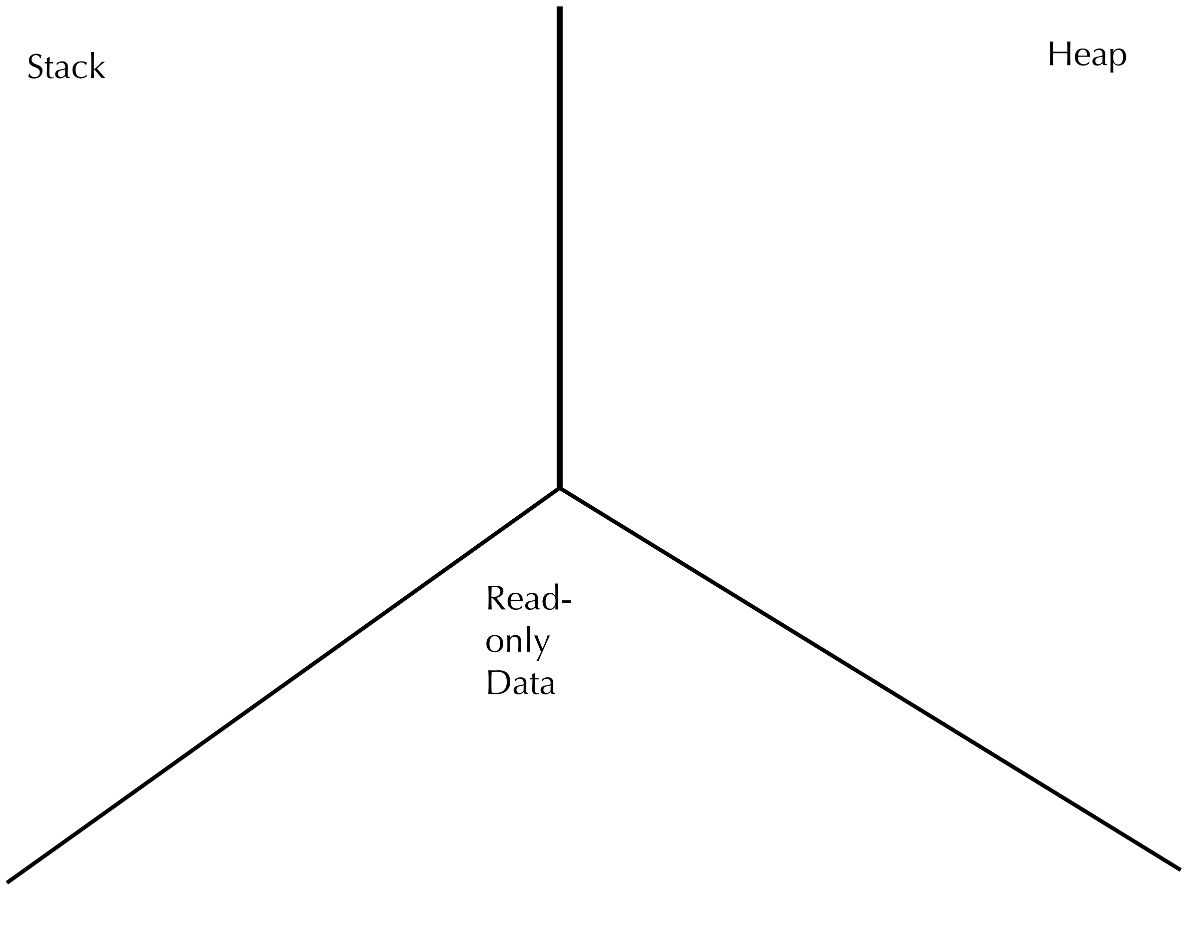
Problem 2: Memory Diagram

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:

```
int eleven = 11;
char *stranger = "things";
int **upside = malloc(3 * sizeof(int*));
upside[0] = malloc(4);
*upside[0] = 2;
upside[1] = &eleven;
upside[2] = (int*) ((char*)stranger + 1);
```

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'.



Problem 3: Strings and pointers (10 points)

(a) (4pts) Consider the following code, compiled using the compiler and settings we have been using for this class.

```
char *str = "Stanford University";
char a = str[1];
char b = *(char*)((int*)str + 3);
char c = str[sizeof(void*)];
```

What are the `char` values of variables `a`, `b`, and `c`? (`a` is filled in for you as an example) Write "ERROR" across the box if the line of code declaring the variable won't compile.

a	`'t'`	b		c	

(b) (6pts) The code below has <u>three buggy lines of code in it</u>. The three buggy parts of the code are noted in bold. <u>Next to each buggy line, write a new line of code that fixes the bug</u>. You may have an idea for restructuring the program that would also fix the bugs, but you must only write code to replace the lines shown in bold—<u>one line of replacement code per one line of buggy code</u>.

The purpose of this function is to take an array of strings (always size 3) and returns a heap-allocated array of size 2, where the first entry is the concatenation of the first two strings in the input array, and the second entry is a copy of the third string in the input array. The two strings in the returned array are <u>both newly allocated on the heap</u>. The input is not modified in any way. You may assume that the input is always valid: the array size is always 3, none of the array entries is NULL, and all strings are valid strings.

```
char **pair_strings(char **three_strings) {
   char *return_array[2];

   ____________________________________________________________;
   size_t str0len = strlen(three_strings[0]);
   return_array[0] = malloc(str0len + strlen(three_strings[1]));

   ____________________________________________________________;
   strcpy(return_array[0], three_strings[0]);
   for (size_t i = 0; i < strlen(three_strings[1]); i++) {


   for (______________________________________________________) {
      return_array[0][str0len + i] = three_strings[1][i];
   }
```

```
    return_array[1] = strdup(three_strings[2]);
    return return_array;
}
```

Problem 4: Bitwise ops

(6pts) As you know, an `unsigned int` on our system is 32 bits or 4 bytes. We can imagine separating the number into bytes and stacking the bytes vertically, like this (for `0xFEFAFE05`):

```
11111110
11111010
11111110
00000101
```

We say a "column" in this view is "odd" if it contains an odd number of 1's. Write a function `odd_cols` that takes an `unsigned int` and returns **`true`** if every column of the input number is odd, otherwise false (i.e., returns false if any column with zero, two, or four 1's in it).

- Example 1: `odd_cols` should return **true** for the input shown above because the rightmost column has three 0's and one 1, and all other columns have three 1's).
- Example 2: return **`false`** for this input: **`0x00000001`**
- Example 3: return **`true`** for this input: **`0x000000FF`**
- For full credit, your code should separate the bytes and then use a small number of bitwise operations that in effect check all 8 columns simultaneously. In particular, your code should not use any loops (evading this restriction with recursion or excessive repetition is not allowed). Correct solutions that do not follow this guideline are worth 4pts (2pt deduction).
- Some code structure is provided to help you get started.
  - The `ptr` variable is intended to help you separate out the rows, but you aren't required to use it.
  - The solution to this problem does not depend on the byte order, so you shouldn't be concerned with little-endian issues, and you may assign the provided `rowN` variables in any order.
  - You are required to fill in the `rowN` variable lines so that the variables collectively hold all four bytes, but after that it's up to you to design your code.

(Write your code on the next page.)

```
bool odd_cols(unsigned int n) {
    unsigned char *ptr = (unsigned char*)&n;
    unsigned char row0, row1, row2, row3;

    row0 = ___________________________________________________;

    row1 = ___________________________________________________;

    row2 = ___________________________________________________;

    row3 = ___________________________________________________;

    // Write the rest of the function here:















}
```

Problem 5: Strings and Pointers Short Answer

(a) (3pts) Write the **<u>decimal</u>** address the pointer arithmetic expression would produce (or say that the pointer arithmetic expression is **illegal**). You should assume that `base` is of type **`void*`** and evaluates to **<u>decimal</u>** 1000. Also assume that this code is compiled and run with our usual class compiler and settings. What address is produced by:

```
(char**) base + 10             _______________

(char*) base + 10              _______________

(int*) base + 10               _______________
```

(b) (4pts) Here is some **<u>buggy</u>** code that takes an array of strings (array of char*) and returns the concatenation of them all together in sequence.

```
char *multi_concatenate(const char *strs[], size_t num_strs) {
     size_t len = 1;
     for (size_t i = 0; i< num_strs; i++) {
          len += sizeof(strs[i]);
     }

     char *result = malloc(len);
     for (size_t i = 0; i< num_strs; i++) {
          memcpy(result + sizeof(strs[i]) * i, strs[i],
               sizeof(strs[i]));
     }
     result[sizeof(strs[0]) * num_strs] = '\0';

     return result;
}
```

Assume the buggy code above is compiled according to our usual class compiler and settings. We call it with the `printf` line below (you may assume that, when run, it manages to not crash at least long enough to complete the `printf` line). Write what is printed by the buggy code above.

```
char *strs[] = {"HELLOWORLD", "IAMREADY", "TOPARTY", "RIGHTNOW!"};
printf("%s\n", multi_concatenate(strs, 4));
```

Output: ____

(c) (3pts) Below is the same **buggy** code from above. **Fix it** so that it correctly takes an array of strings (array of char*) and returns the concatenation of them all together in sequence. You may cross out code, add code, or edit code, as needed (just please be as neat and clear as possible, thank you! ☺).

```
char *multi_concatenate(const char *strs[], size_t num_strs) {

    size_t len = 1;

    for (size_t i = 0; i< num_strs; i++) {

        len += sizeof(strs[i]);

    }

    char *result = malloc(len);

    for (size_t i = 0; i< num_strs; i++) {

        memcpy(result + sizeof(strs[i]) * i, strs[i],

            sizeof(strs[i]));

    }

    result[sizeof(strs[0]) * num_strs] = '\0';

    return result;

}
```

Problem 6: The `accumulate` generic

Consider the following two functions, noticing that they have very similar structure:

```
int int_array_product(const int array[], size_t n) {
  int result = 1;
  for (size_t i = 0; i < n; i++) {
    result = result * array[i];
  }
  return result;
}

double double_array_sum(const double array[], size_t n) {
  double result = 0.0;
  for (size_t i = 0; i < n; i++) {
    result = result + array[i];
  }
  return result;
}
```

You are to implement a generic **`accumulate`** function that captures the shared structure of these two funcions. It takes the base address of an array and its effective length, the array element size, the function callback that should be repeatedly applied (above it would be multiply and add, but implemented as 2-argument functions rather than operators directly), the address of the default/starting value (to play the role of 1 and 0.0 in the above code), and the address where the overall result should be placed. The function type of the callback is as follows:

```
typedef void (*BinaryFunc)(void *partial, const void *next, void *result);
```

Any function that can interpret the data at the first two addresses, combine them, and place the result at the address identified via the third address falls into the **`BinaryFunc`** function class. (The **`const`** appears with the second address, because it's expected that the array elements—the elements that can't be modified—be passed by address through the **`next`** parameter.)

a) First, implement the generic **`accumulate`** routine. We've provided some of the structure that should contribute to your implementation. You should fill in the three arguments needed so that **`memcpy`** can set the space addressed by result to be identical to the space addressed by **`init`**, and then go on to fill in the body of the **`for`** loop.

   *This first part was designed to expose basic memory and pointer errors very early on—e.g. to confirm that weren't dropping **`&`**'s and **`*`**'s where they weren't needed.*

```
void accumulate(const void *base, size_t n, size_t elem_size,
                BinaryFunc fn, const void *init, void *result) {
```

b) Now reimplement the **`int_array_product`** function from the previous page to leverage the **`accumulate`** function you just implemented for part a). Assume the name of the callback function passed to **`accumulate`** (which you must implement) is called **`multiply_two_numbers`**.

```
static void multiply_two_numbers(void *partial, const void *next, void
*result) {
```

```
int int_array_product(const int array[], size_t n) {
```

Problem 7: Integer Representation

a) Write the unsigned binary number 101100111010 in hexadecimal:

b) Write the signed (two's complement) binary number 101101 in decimal:

c) Write the signed (two's complement) binary number 001101 in decimal:

d) Write the hexadecimal number 0x54BEEF as an unsigned binary number:

e) Write the decimal number 28 as an 8-bit binary number:

f) Write the decimal number 28 in hexadecimal:

g) Write the decimal number -33 as an 8-bit signed (two's complement) binary number:

Problem 8: Integer Representation

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 on the provided lines*.

a) (2pts) Write the unsigned binary number `1100101011111110` in hexadecimal:

____

b) (2pts) Write the decimal number `55` as an 8-bit signed (two's complement) number:

____

c) (2pts) Write the 8-bit signed (two's complement) hexadecimal number `0xAA` in decimal:

____

d) (2pts) Write the decimal number -13 in 8-bit signed (two's complement):

____

Problem 9: Strings and pointers

(c) (6pts) Consider the following code, compiled using the compiler and settings we have been using for this class.

```
char *str = strdup("ABCDEFGHIJKLMNOPQRSTUVWXYZ");
char a = str[0];
char c = *(char *)((int *)str + 2);
str++;
char d = str[3];
```

Note: recall that type `short` is like `int` but 2 bytes.

What are the `char` values of variables `a`, `b`, `c`, and `d`? (`a` is filled in for you as an example) Write "ERROR" across the box if the line of code declaring the variable won't compile, or executing the operation could give a memory error in Valgrind (e.g., off end of array).

`a`	`'A'`	`c`	
`d`			

A generic `CMap` type maps string keys to arbitrary values. It works with the following functions:

```
// returns the first key in the map as a char *,
// or NULL if there are no keys
char *cmap_first(CMap *cmap);

// returns a pointer to the value associated with that key, or NULL
// if the key does not exist.
void *cmap_get(cmap, char *key);

// returns the next char * key in the map following this key,
// or NULL if there are no more keys.
char *cmap_next(cmap, char *key);

// returns the number of elements (key/value pairs) in the map.
int cmap_count(cmap);
```

(d) (6pts) The code below has <u>three buggy lines of code in it</u>. The buggy parts of the code are noted in bold. The purpose of this function is to take a pointer to a `CMap` that maps each key (a string, as usual for `CMap`) to a value of type `int`. The function is supposed to put the map's values in an array of integers (order of the values does not matter), and also make a copy of the first key (the key that comes back from `cmap_first`). The copy of the first key and the array are to be "returned" to the caller by setting two parameters that are pointers passed "by reference."

- The copy of the first key should be a <u>new heap copy</u>.
- The array of integer values should be <u>created on the heap</u>.
- The input parameters `first_key` and `values` are pointers passed "by reference" that should be set to point to a new heap copy of the first key, and an array holding the map's values, respectively.
- Be VERY careful about order of operations with * and []—you are encouraged to overuse parentheses to make your meaning clear and avoid mistakes with this.
- Note that there <u>may be more than one distinct bug</u> per buggy line.

<u>Below each buggy line, write a new line of code that fixes the bug</u>. You may have a different idea for restructuring the program that would also fix the bugs, but you must only write code to replace the lines shown in bold—<u>one line of replacement code per one line of buggy code</u>.

```
void separate_map(CMap *cmap, char **first_key, int **values) {
   first_key = cmap_first(cmap);

   ____________________________________________________________;
   int nelems = cmap_count(cmap);
   values = malloc(nelems);

   ____________________________________________________________;
   int index = 0;
   for (const char *cur = cmap_first(cmap); cur != NULL;
                    cur = cmap_next(cmap, cur)) {
      values[index] = cur;

    ___________________________________________________________;
    }
 }
```

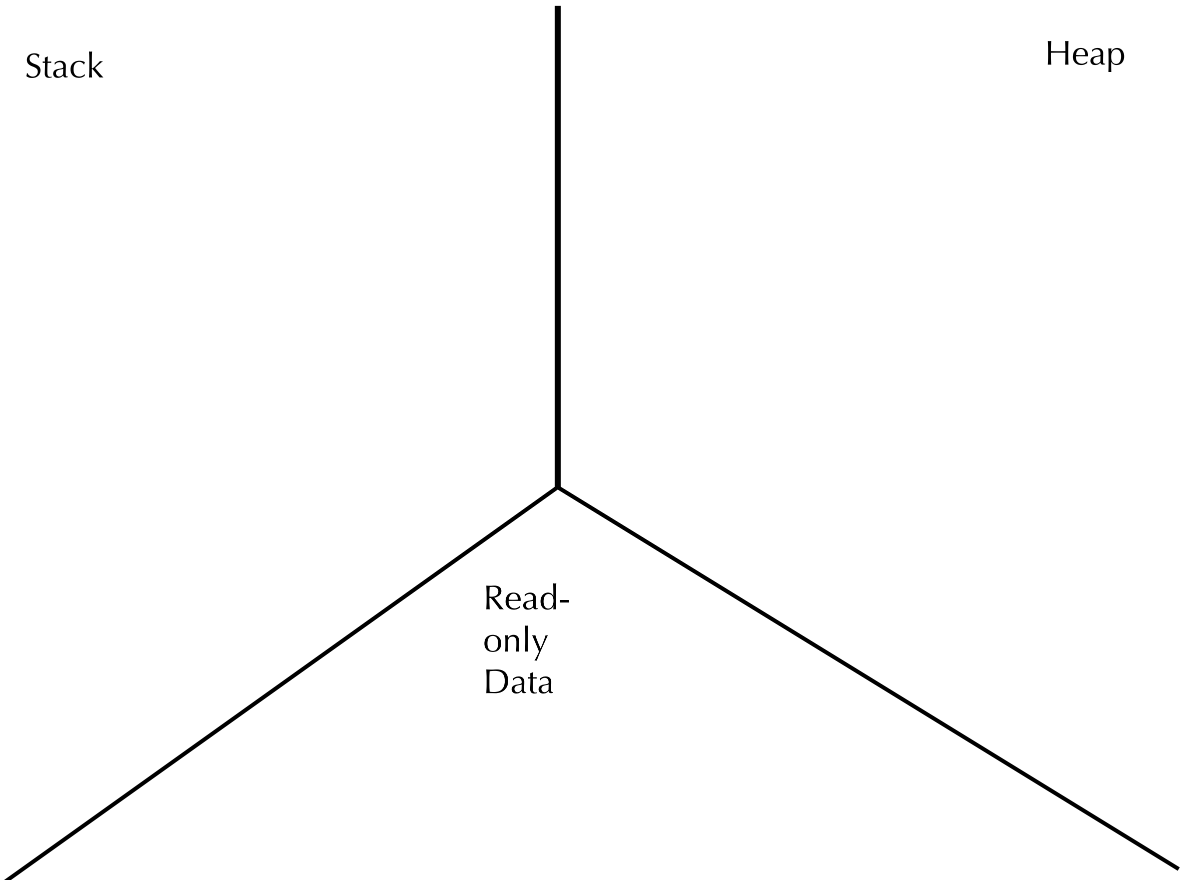
Problem 10: Memory Diagram

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 *tree = "STANFORD";
    char beat[8];
    char **fountain = malloc(8);
    int *leland[2];
    strcpy(beat, "Cal");
    fountain[0] = strdup("hop");
    leland[0] = malloc(4);
    *(leland[0]) = 5;
    leland[1] = leland[0];
```

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'`.


Stack
Heap
Read-only Data