CSM 61C Spring 2020

C Memory

Exam Question Compilation Solutions

This document is a PDF version of old exam questions by topic, ordered from least difficulty to greatest difficulty.

Questions:

- Fall 2015 Final MT1-1B
- Spring 2015 Final M1-1C
- Fall 2018 Quest Q3A
- Summer 2018 Midterm 1 Q2
- Fall 2019 Quest Q5
- Summer 2019 Midterm 1 Q2
- Fall 2018 Final M2 Part 2
- Spring 2018 Midterm 1 Q3
- Summer 2018 Final Q2

SID: _____

MT1-1: Potpourri - Good for the beginning… (8 points)

i. False. The compiler turns C code into assembly code, not things to be passed into processor just yet
ii. False. The immediate needs to be broken up as it is too long. Also, register names still need to be converted.
iii. False. This is done in the Assembler (pass 1/pass 2)

b. **Memory Management**

```
int global = 0;

int* func() {
    int* arr = malloc(10 * sizeof(int));
    return arr;
}

int main() {
    char* str = "hello world";
    char str2[100] = "cs61c";
    int* a = func();
    return 0;
}
```

In what part of memory are each of the following values stored?

***str: static**
str2[0]: stack
a: stack
arr: stack
arr[0]: heap

*str: Static. "hello world" is a string literal and string literals are stored in Static Data. Because this is a pointer and not an array, the pointer points to actual string literal. Thus, dereferencing the pointer leads to a value stored in Static Data.
str2[0]: Stack. Although "cs61c" is a string literal, because we are assigning into an array and the array is a separate chunk of memory stored on the Stack, the string literal gets copied into str2. Thus, str2[0] exists on the Stack.
a: Stack. We are declaring an integer pointer called a, which is stored on the stack.
arr: Stack. arr is created in a function, thus is on the Stack.
arr[0]: Heap. arr itself is stored on the stack, but we allocated memory in the Heap, which is what arr points to. So dereferencing arr yields a value stored in the Heap.

in 2015 Final l i n

SID: _____

M1-1: *I smell a potpourri section covering midterm one…* (9 points)

c) Consider the C code below. Indicate where the values on the right live in memory (using **(S)**tack, **(H)**eap, s**(T)**atic, or **(C)**ode). Assume no registers are used:

```
#define a 10
int b = 0;

int main(int argc, char** argv) {
   int c = a;
   char d[10];
   int* e = malloc(sizeof(int));
}
```

Explination on next page

a: C

b: T

***d:** S

***e:** H

e: S

in 2015 Final l i n n

1c)

a → this is a macro, so the symbol "a" is replaced by 10 by the compiler. Thus, "a" is stored in the code section.

b → this is a global variable, so it is stored on static

*d → d is an array that is locally allocated (within the "main" function), so d refers to a block of memory on the stack

*e → e is a local variable, but its value is an address on the heap. Thus, when we dereference e, the block of memory e refers to is on the heap.

e → e itself is a local variable, so it is located on the stack

```
// My project partner wrote this code to duplicate some elements of orig into copy
int orig[] = {1,2,3,4,5,6,7,8}; // ints are 4 bytes wide
int main() {
    int *backup, *copy, **copyH;
    backup = copy = (int *) malloc (sizeof(int) * 100);
    copyH = &copy;
    for (int i = 0; i < 2; i++) {
        *copy = orig[i];
        *copyH = *copyH + 4;
    }
}
```

Q3a) Right *before* the **for** loop, where in memory do the following **point**? (Select ONE per row)

	Code	*Static*	*Stack*	*Heap*
orig	○	●	○	○
backup	○	○	○	●
copyH	○	○	●	○

Question 2: C Memory Management (16 pts)

```
char *mood;
char *copy_message (char *msg) {
     char *x = malloc (sizeof (char) * (strlen (msg) + 1));
     strncpy (x, msg, strlen (msg));
     x[strlen (x)] = '/0';        / **** 6 ****/
     return x;
}
void print_int (int *p) {
     printf ("%d\n", *p);                /**** 7 ****/
}
void print_msg (char *str) {
     char *cpy = calloc (strlen (str) + 1, 1);
     strncpy (cpy, str, strlen (str));
     printf ("%s\n", cpy);          /**** 8 ****/
}
char *a () {
     char res[7] = " rules";
     return res;
}
char *b () {
     char *var = "cs 61c";
     return var;
}
void c () {
     printf ("%s\n", a ());       /**** 9 ****/
     printf ("%s\n", b ());      /**** 10 ****/
}
int main () {
     int y;
     mood = malloc (3);
     strcpy (mood, "hi");
     copy_message (mood);
     print_int (&y);
     print_msg (mood);
     c ();
}
```

Each of the following values below evaluates to an address in the C code on the previous page. Select the region of memory that the address points to (notice each function is called exactly once).

1. `mood`	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap
2. `&mood`	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap
3. `var`	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap
4. `res`	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap
5. `print_int`	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap

On the previous page there are comments on lines with numbers from 7-11. Each of these refers to a line of code that requires a dereference of a pointer to be performed. What we want to do is characterize if these memory accesses are legal c. We will use the following terminology

Legal: All addresses dereferenced are addresses that the program is allowed to read.

Initialized: Is there actual meaningful data in contents (data at each address) or is it garbage.

Always Illegal: This line will always dereference an address the program doesn't have explicit access to

Possibly Legal: The operation could result in only dereferences of legal addresses but it's also possible that in other runs on the program illegal accesses occur.

For each of lines that have the numbered comment select the best answer from

- A. Legal and Initialized
- B. Legal and Uninitialized
- C. Possibly Legal
- D. Illegal

For example for question 6 you should answer about the line with the /**** 6 ****/ comment from when the program runs.

6.	Ⓐ	Ⓑ	Ⓒ	Ⓓ
7.	Ⓐ	Ⓑ	Ⓒ	Ⓓ
8.	Ⓐ	Ⓑ	Ⓒ	Ⓓ

9. Ⓐ Ⓒ

10. Ⓐ Ⓒ

Fall 2019 Quest Solution

Q5) [10 Points] 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*
`arr`	○	●	○	○
`arr[0]`	○	●	○	○
`dest`	○	○	●	○
`dest[0]`	○	○	○	●
`&arrPtr`	○	○	●	○

- `arr` is an array of character pointers, so it itself is a pointer to the first element in the array. Since the array is declared globally, its contents are placed in the static portion of memory, so `arr` points to **static**.
- `arr[0]` is a string literal (therefore, a pointer to the first character in the string) so it points to the static portion of memory.
- dest is an array declared in the main function, so all of the elements in dest are in the stack.
- Since dest is a pointer to the first element, it points to the stack.
- Here, we notice the line where we set dest [i] to be the result of a malloc. Since malloc allocates space in the heap, dest[0] is a pointer into the heap.
- arrPtr is declared at the top of main, so it is in the stack. The address of arrPtr is therefore a pointer into the stack.

Summer 2019 Midterm 1 Solution

Question 2: Remember remember the segments of memory. - 14 pts

```
#include <stdlib.h>
#include <stdbool.h>

bool fetch_data (char* buf);

char* receive_buffer;
bool is_complete = false;

int main(int argc, char* argv[]) {
      receive_buffer = malloc(100*sizeof(char));
      if (!receive_buffer) {
            return -1;
      }
      fetch_data(receive_buffer);
      free(receive_buffer);
      return 0;
}

/* Function that takes in a buffer for storing characters
 * and places data in the buffer by calling receive_data.*/
void fetch_data(char* buf) {
      int len = receive_data(receive_buffer);
      if (len == 0) {
            return; // HERE
      } else {
            fetch_data(buf + len);
      }
}
```

All of the following expressions **evaluate** to an address value. State in which region of memory each value corresponds to: stack, heap, static, or code. Assume we are about to execute the line marked HERE.

1.	receive_buffer	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap (set by malloc(100…))
2.	&(receive_buffer[0])	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap (same as question 1)
3.	&receive_buffer	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap (is a global variable)
4.	&argc	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap (arguments are local on stack)
5.	&is_complete	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap (is a global variable)
6.	&fetch_data	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap (function pointers in code)
7.	buf	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap (buf was receive_buffer passed in, so this is the same as question 1)

Summer 2019 Midterm 1 (cont'd)

Now consider the following different program:

```
// function prototypes
int a(int x);
int b(int y);

int main(void) {
      a(3);
      return 0;
}

int a(int x) {
      return b(x - 1);
}

int b(int y) {
      if (y <= 1) {
            return 7; // HERE
      } else {
            return a(y - 1);
      }
}
```

Assume we are about to execute the line marked HERE. Label all the stack frames below with either the function that created the frame: a, b, or main, or with UNUSED if the frame is not in use.

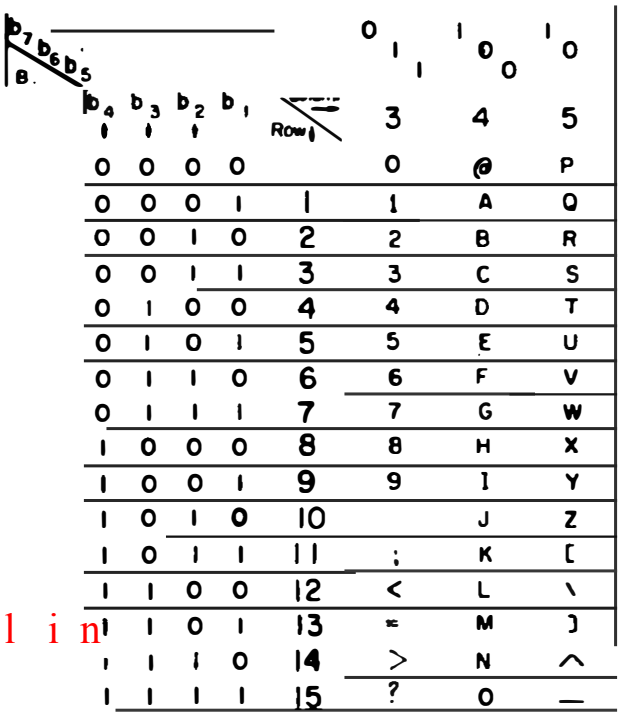
Main calls A(3) calls B(2) calls A(1) calls B(0) which hits the `return 7` line

Memory (Top)					
1	1.	Ⓐ a	Ⓑ b	Ⓒ main	Ⓓ UNUSED
2	2.	Ⓐ a	Ⓑ b	Ⓒ main	Ⓓ UNUSED
3	3.	Ⓐ a	Ⓑ b	Ⓒ main	Ⓓ UNUSED
4	4.	Ⓐ a	Ⓑ b	Ⓒ main	Ⓓ UNUSED
5	5.	Ⓐ a	Ⓑ b	Ⓒ main	Ⓓ UNUSED
6	6.	Ⓐ a	Ⓑ b	Ⓒ main	Ⓓ UNUSED
7	7.	Ⓐ a	Ⓑ b	Ⓒ main	Ⓓ UNUSED

Fall 2018 Final Solution

Consider the following code (and truncated ASCII table; as an example of how to read it, "G" is `0b0100 0111`):

```
uint8_t mystery (uint8_t *A) {
   return *A ? (*A & mystery(A+1))   0xFF;
}
```



Row	3	4	5
0	0	@	P
1	1	A	Q
2	2	B	R
3	3	C	S
4	4	D	T
5	5	E	U
6	6	F	V
7	7	G	W
8	8	H	X
9	9	I	Y
10		J	Z
11	;	K	[
12	<	L	\
13	=	M	]
14	>	N	^
15	?	O	_

d) Where is **A** stored? (*not* what it points to, ***A**)

◯code ◯static ◯heap ◯stack

stack (since it's a local variable / parameter / immediate)

e) What is **(char)mystery("GROW")**? _____

The code does an AND of all the characters bits, the upper bits are 100 & 101 → 100, and the lower bits are 1111 & 0111 & 0010 → 0010, so it's 100 0010 → **B**

e) [alternate exam]

What is **(char)mystery("FLOW")**? _____

The code does an AND of all the characters bits, the upper bits are 100 & 101 → 100, and the lower bits are 1111 & 0111 & 0110 & 1100 → 0100, so it's 100 0010 → **D**

f) A two-character string is passed into **mystery** that makes it return the **uint8_t** value **0** (not the character "**0**"). The first character is "**M**" ["**K**" alternate exam], the second character is a number from 1-9. Which?

◯1 ◯2 ◯3
◯4 ◯5 ◯6
◯7 ◯8 ◯9

What number has no bits in common with M's bits=100 1101 → all numbers have the high nibble with no bits in common so it's only the bits that only have 1 in the 2s column, thus 0010 or 0000 (but 0 is not part of it), so it must be 0010 → **2**. (note the ASCII low nibble of a 0-9 number is the number itself)

[Alternate exam] What number has no bits in common with K's bits=100 1011 → all numbers have the high nibble with no bits in common so it's only the bits that only have 1 in the 4s column, thus 0100 or 0000 (but 0 is not part of it), so it must be 0100 → **4**. (note the ASCII low nibble of a 0-9 number is the number itself)

Problem 3 ***C Analysis*** **(10 points)**

The CS61C Staff is creating songs in preparation of the grading party. Consider the following program:

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

typedef struct Song {
     char *title;
     char *artist;
} Song;

Song * createSong() {
     Song* song = (Song*) malloc(sizeof(Song));
     song->title = "this old dog";
     char artist[100] = "mac demarco";
     song->artist = artist;
     return song;
}


int main(int argc, char **argv) {
     Song *song1 = createSong();
     printf("%s\n", "Song written:");
     printf("%s\n", song1->title); // print statement #1
     printf("%s\n", song1->artist); // print statement #2

     Song song2;
     song2.title = malloc(sizeof(char)*100);
     strcpy(song2.title, song1->title);
     song2.artist = "MAC DEMARCO";
     printf("%s\n", "Song written:");
     printf("%s\n", song2.title); // print statement #3
     printf("%s\n", song2.artist); // print statement #4

     return 0;
}
```

(a) What type of address does each value evaluate to? Fill in the entire bubble.

i. `song1`

- ○ Stack address
- ○ Static address
- ● Heap address
- ○ Code address

ii. `song1->title`

- ○ Stack address
- ● Static address
- ○ Heap address
- ○ Code address

iii. `song1->artist`

- ● Stack address
- ○ Static address
- ○ Heap address
- ○ Code address

iv. `&song2`

- ● Stack address
- ○ Static address
- ○ Heap address
- ○ Code address

v. `song2.title`

- ○ Stack address
- ○ Static address
- ○ Heap address
- ○ Code address

(b) Will all of the print statements execute as expected?

- ○ Yes
- ● No

If you answered yes, leave this blank. If you answered no, write the number(s) of the print statement(s) which will not execute as expected.

Solution: print statement #2

SID: _____

1. `main`	Ⓐ Stack	Ⓑ Heap	Ⓒ Static	Ⓓ Code
2. `&sp`	Ⓐ Stack	Ⓑ Heap	Ⓒ Static	Ⓓ Code
3. `sp`	Ⓐ Stack	Ⓑ Heap	Ⓒ Static	Ⓓ Code
4. `*sp`	Ⓐ Stack	Ⓑ Heap	Ⓒ Static	Ⓓ Code
5. `sp->func_name`	Ⓐ Stack	Ⓑ Heap	Ⓒ Static	Ⓓ Code
6. `&fork`	Ⓐ Stack	Ⓑ Heap	Ⓒ Static	Ⓓ Code
7. `argv`	Ⓐ Stack	Ⓑ Heap	Ⓒ Static	Ⓓ Code

Suppose we had a simple recursive function defined as follows:

```
long factorial(long n):
    if (n == 1):
        return 1;
    else:
        return n * factorial(n-1)
```

Assume that function frames only require space for the local variables (i.e. the return value of `frame_size("factorial")`). You are given the following specifications:

Stack and Heap: 16 KiB
Static: 12 KiB
Code: 4 KiB

`frame_size("factorial") = 8`
`sizeof(StackNode) = 56`

Suppose we call the `factorial` function on some number N **using our Stack data structure from Question 1** (note: we allocated data for our `StackNode` structs):

```
StackNode *sp = NULL;
int main () {
      push("factorial", &sp);
      push("factorial", &sp);
      push("factorial", &sp);
      ...

      // the N'th call to factorial
      push("factorial", &sp);

      // the first return from factorial
      pop(&sp);
      ...
}
```

What is the smallest value of N that will cause a maximum recursion depth error (meaning no more function frames can be created)? Ignore the stack space required for the `main` function. If convenient, put your answer as a power of 2.

2^{14} B of Heap Space; Each function call needs 8 + 56 = 64 = 2^6 B of heap space

N = $2^{14} / 2^6 = 2^8$