Midterm Exam Spring 2024

This exam consists of 4 substantive questions. You have 50 minutes – budget your time wisely. Assume the ArmLab/gcc217 environment unless otherwise stated in a problem.

Do all of your work on these pages. You may use the provided blank spaces for scratch space, however this exam is preprocessed by computer, so for your final answers to be scored you must write them inside the designated spaces and fill in selected circles and boxes completely (● and ■, not ✔ or ✘). Please make text answers dark and neat.

Name: **Sample Solutions** NetID:

Precept:

○ P01 / P02 - MW Xiaoyan Li	○ P10 - TTh 12:30 Dwaha Daud	○ P07 TTh 2:30 Nanqinqin Li
○ P03 - TTh 12:30 Donna Gabai	○ P05 - TTh 1:30 Donna Gabai	○ P08 TTh 3:30 Indu Panigrahi
○ P04 - TTh 12:30 Guðni Nathan Gunnarsson	○ P06 - TTh 1:30 Austin Li	○ P09 TTh 7:30 Gongqi Huang

This is a closed-book, closed-note exam, except you are allowed one one-sided study sheet. Please place items that you will not need out of view in your bag or under your working space at this time. Electronic devices such as cell phones, laptops, smartwatches except to check the time, etc. may not be used during this exam.

This examination is administered under the Princeton University Honor Code. Students should sit one seat apart from each other and refrain from talking to other students during the exam. All suspected violations of the Honor Code must be reported to honor@princeton.edu.

In the box below, copy **and** sign the Honor Code pledge before turning in your exam:
"I pledge my honor that I have not violated the Honor Code during this examination."

Exam Statistics:
Max: 50/50
Median: 36/50
Mean: 34.29/50
Standard Deviation: 8.08/50 X____________________________

Question 0: *Logistics* 0 points

Please don't make the course staff's life harder: make sure you have filled out the fields for your name, your NetID (not PUID, not email alias), precept and the Honor Code pledge text on the front page. Sign your name once you have finished the exam.

Question 1: *Half As Interesting* as the subsequent questions? 12 points

a. What are the values of the variables in this code after the loop terminates?

```
int x = 0;
int y = 0;
while(x++ < 3)
   y += ++x;
```

x: **5**

y: **6**

*** Initially x and y are each 0.**
*** In the 1st check, x becomes 1, but x++ emits the old value, so the check is 0 < 3 (true)**
*** in the 1st body iteration, x becomes 2, which is added to y, so y also becomes 2.**
*** In the 2nd check, x becomes 3, emits the old value 2, so 2 < 3 (true)**
*** In the 2nd body iteration, x becomes 4, which is added to y, so y becomes 6**
*** In the 3rd check, x becomes 5, emits old value, so 4 < 3 (false)**
*** Thus, after the loop, x:5, y:6.**

b. What is the base 10 value of the variable in this code after the assignment?

```
int z = (6 << 5) | (6 << 2) | -~0;
```

z: **217**

```
       6 is: 0...000000110
  6 << 5 is: 0...011000000   ***
  6 << 2 is: 0...000011000   ***
       0 is: 0...000000000
      ~0 is: 1...111111111
   -(~0) is: 0...000000001   ***
   using 2's complement algorithm

Bitwise-OR of 3 ***'ed partial
results above yields:
0...011011001

Convert to base 10:
128 + 64 + 16 + 8 + 1 = 217
```

c. In at most 10 words, describe what the *`Amystery`* function does (i.e., for what property of its parameters does it return `1`):

```
int Amystery(int i1, int i2) { return (i1 ^ i2) < 0; }
```

This function returns whether its two parameters' signs are different. (i.e., returns true iff one parameter is negative and one is non-negative).

^ is the exclusive or bitwise operator, which results in a 0 if the corresponding bits from the two operands are the same (either both 0 or both 1) and a 1 if the corresponding bits are different (one of each).

Checking < 0 is a check for a negative number, which in 2's complement representation means the number has a 1 as its leftmost bit.

Thus, in order to return true, the exclusive or of the two parameters' leftmost bits must be 1, and thus the two parameters' leftmost bits must be different.

Question 2: We *train*ed for this: we *Wendover* `gcc`'s stages a lot 6 points

Consider the following C code:

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

enum {N = 10};

/* print the contents of arr, separated by tabs */
void printarr(int arr[N]) {
   size_t i;
   for(i = 0; i < (size_t) N; i++) {
      printf("%d\t", arr[i]);
   }
}
```

a. Which of the 4 build stages processes the line `#include <stdio.h>`?

Preprocessor ● ○ Compiler
Assembler ○ ○ Linker

Constructs starting with # are preprocessor directives.

b. Which line is **not** likely added to the code when processing `#include <stdio.h>`?

`extern int getchar (void);` ○ ○ `#define _STDIO_H 1`
`#define EOF (-1)` ○ ● `void printarr(int arr[N]);`

The stdio.h header would have a function declaration for getchar, a macro definition of EOF, and a guard against multiple inclusion, but NOT a declaration of a function we defined in our code.

c. Which is the first stage that would produce a warning or error if the line `#include <stdio.h>` were omitted?

Preprocessor ○ ● Compiler
Assembler ○ ○ Linker

Without the declaration of printf, the COMPILER cannot type-check a function call to printf, so it emits a warning saying that it hasn't seen a declaration for the function ("implicit declaration").

d. The function `printarr` has a major problem that may result in behavior that is not expected by the client. Describe the issue in at most 2 sentences.

The function body assumes that the argument must be an array of size N (perhaps because of the function parameter arr[N]. But this is not enforced by C: a caller can pass an array of any size into the function (see revfn2.c from precept w04p1)

So if a caller passes an array that has more than 10 elements, the elements beyond index 9 will not be printed, despite the claim of the function's comment. And if a caller passes an array that has fewer than 10 elements, the loop will traverse out of bounds off the end of the array.

Question 3: ~~*Jet*~~ *Set Lag: The Game* 14 points

Consider the following plausible addition to your String library from Assignment 2:

```
/* Change the char at existing index i of pc to be c */
void Str_set(char *pc, size_t i, char c) {
   assert(pc != NULL);
   if(i < Str_getLength(pc))
      pc[i] = c;
}
```

Further consider some client code using this function – assume these lines appear inside a function body, that all necessary headers have been `#include`d, and that all required memory cleanup occurs later in the function:

```
char ac[] = "Denby";
char *temp;
char *pc1 = "The Boys";
char *pc2 = (char *) malloc(8);

Str_set(ac, 0, 'H');
Str_set(ac, 3, 'd');

Str_set(pc1, 4, 'G');
Str_set(pc1, 5, 'u');

if(pc2 != NULL) {
   Str_copy(pc2, "McManus");
   temp = Str_search(pc2, "M") + 2;
   if(temp != NULL) Str_copy(temp, "Khare");
   Str_set(pc2, 1, '.');
}
```

a. Rewrite the assignment `pc[i] = c;` on the last line of `Str_set`'s body using equivalent pointer notation instead of array indexing bracket notation:

*(pc + i) = c;

b. The argument 8 to `malloc` is the correct amount to request for this program on armlab. Why is 8 the correct number of bytes to allocate?

○ 8 is the result of `Str_getLength("McManus")` on armlab
● 8 is the result of **`sizeof`**`("McManus")` on armlab
○ 8 is the result of **`sizeof`**`(`**`char`**` *)` on armlab

We must request enough space to store all the elements of the array, **including the '\0' that ends the string.**

Str_getLength returns the number of chars **strictly before the ‘\0’.**

char * is the type of pc2, but has nothing to do with how much memory must be dynamically allocated for the string.

Each row in the table below considers a variable from the client code on page 4. You need **not** answer the grayed out cells. (A bare array name is often treated as a pointer to the 0th array element, but is not actually a pointer variable; `temp` has no `Str_set` call.)

c. In the LOCATION column, indicate in which memory section the variable resides. If the variable's declaration results in a compile-time error, write “ERROR”.

d. In the TARGET column, indicate which memory section that variable *references* (i.e., where does it point) after the four declarations at the top of the code.

If there is not enough information to know at that point, or if it is not deterministic, write “NEI”. If the variable's declaration does not compile, write “ERROR”.

e. In the CONTENTS column, indicate the string's contents through the first `'\0'`, as they are in memory immediately **after** the `Str_set` call(s) for that string.

Assume for the last row in this column that execution <u>does</u> reach the call for `pc2`. If a call results in contents that are nondeterministic, indicate this with “NEI”. If a call results in a runtime crash, indicate this with “ERROR”.

LOCATION:
All three of temp, pc1, pc2 are local vars: stored in their function's stackframe.

TARGET:
ac is an array local var so all its elements are on the stack. ac implicitly references ac[0].

temp is an uninitialized local var, so its initial contents are not well defined.

pc1 points at a string literal, which is a const char array in RODATA.

If malloc succeeds, pc2 will point to the heap. pc2 will be NULL if malloc fails. The problem does not indicate to assume malloc's success for this column.

	LOCATION	TARGET	CONTENTS
`ac`		**STACK**	**{‘H’, ‘e’, ’n’, ‘d’, ‘y’, ‘\0’}**
`temp`	**STACK**	**NEI**	
`pc1`	**STACK**	**RODATA**	**ERROR**
`pc2`	**STACK**	**NEI**	**{‘M’, ‘.’, ’K’, ‘h’, ‘a’, 'r', 'e', ‘\0’}**

CONTENTS:
Initially, ac is {‘D’, ‘e’, ’n’, ‘b’, ‘y’, ‘\0’}.The first call changes ‘D’ to ‘H’. The second call changes ‘b’ to ‘d’.

The string contents “The Boys” are in RODATA, so they cannot be changed. Trying to do so is a runtime error.

For pc2 to reach the Str_set call, which is assumed for this column, malloc must have succeeded. Initially after malloc, the memory is uninitialized. The first Str_copy call sets it to be {‘M’, ‘c’, ‘M’, ‘a’, ‘n’, ‘u’, ‘s’, ‘\0’}. The Str_search call returns the address of the first ‘M’, so that pointer + 2 will result in the address of the second ‘M’. The — improperly guarded :(— second Str_copy call replaces “Manus” with “Khare”. The Str_set call changes the ‘c’ to ‘.’.

Question 4: Memory Management *Crime Spree*

18 points

Consider the following C code, which you may assume `#include`s all required `.h` files:

```c
struct S {
   size_t ulCount;
   int *pi;
};

enum {N = 9};

/* set ps's pi field to a new array {1, 2, ..., 9}
   whose memory will be owned by the caller, and
   set ps’s ulCount field to be the array’s length */
void S_initSeasons(struct S *ps) {
   size_t ulIndex;
   free(ps->pi);
   (*ps).pi = malloc(N * sizeof(int));
   for(ulIndex = 1; ulIndex <= (size_t) N; ulIndex++)
      (ps->pi)[ulIndex] += ulIndex;
   (*ps).ulCount = N;
}
```

a. The function `S_initSeasons` has *many* problems. In the box below, describe **3** bugs, each in no more than 10 words. Only the first 3 bugs listed will be graded.

   A bug in this problem is something that may:

   - ○ cause a compiler warning or error
   - ○ cause a runtime crash
   - ○ constitute a memory management error
   - ○ violate the specified behavior from the function’s comment

These were bugs, as defined by the problem:
*** does not validate that ps is not a NULL pointer (will segfault when dereferenced if NULL)**
*** frees ps->pi, which might not have been initialized or might not be dynamically allocated.**
*** does not check the return value from malloc (will segfault when dereferenced if NULL)**
*** fills the array starting at index 1 (skips element 0, shifts elements by 1, accesses OOB element 9)**
*** uses uninitialized values in setting the contents. malloc doesn’t initialize the memory it allocates, so using += adds the season number to a junk value.**

These were "non-bugs", i.e. plausible-but-incorrect answers:
*** does not free the memory it mallocs. This isn’t a leak, as the function’s purpose is to assign the pi field a value that can be used after return. (Contract says the memory will be owned by caller.)**
*** doesn't cast the pointer returned by malloc: void * may be assigned without a cast to any ptr type.**
*** does have a dangling pointer, ps->pi (after having freed it unnecessarily — see bugs above), but simply having a dangling pointer isn't a memory management bug, so long as that pointer isn’t ever dereferenced, which it isn’t until after it is pointed to a different allocation.**
*** swaps back and forth between */. and -> access to the structure's fields, and uses array access notation instead of pointer notation. In both cases, the two options are equivalent and valid.**
*** mixes int-vs-size_t, but given the small fixed values in the code, this will not be a problem**
*** ps is a pointer to a struct, not a struct copy itself, so changes **will** be made to the caller’s struct**

Now consider a second function that uses the **`struct S`** definition from page 6:

```
void S_addSeason(struct S *ps) {
   size_t ulNew;
   assert(ps != NULL);

   ulNew = ps->ulCount + 1;

   /* insert code here */

   ps->pi[ps->ulCount] = ulNew;
   ps->ulCount = ulNew;
}
```

This function updates a structure initialized by a previous call to (a correct version of) **`S_initSeasons`**. Consider these three possible completions for the function at the location designated by the comment in the code:

b. `realloc(ps->pi, ulNew * sizeof(int));`

c. `ps->pi = realloc(ps->pi, ulNew * sizeof(int));`

d.
```
{
   int *pi; /* declaration at the top of an inner block is okay */
   pi = realloc(ps->pi, ulNew * sizeof(int));
   if(pi != NULL) ps->pi = pi;
}
```

In the table below, each row specifies a potential result of the `realloc` call. For each potential result, indicate whether each of the three possible completions would end up with a correct update to `ps` ("OK") or would end up in a state that would lead to a memory error ("BAD").

	b.		c.		d.	
	OK	BAD	OK	BAD	OK	BAD
Succeeds without relocation	●	○	●	○	●	○
Succeeds but relocates	○	●	●	○	●	○
Fails	○	●	○	●	○	●

See discussion at top of page 8.

(Question 4 was the last question – enjoy your *mandatory rest period* … er … spring break! This page is intentionally left blank. Its *Extremities* may be used for scratch work from any problem, however any answers given on this page will not be graded. If you finish the exam early, feel free to use this space to sketch a *nebula* or write a treatise extolling the virtues of *a local snack* from your hometown.)

When realloc expands without moving the allocation, the realloc call returns the same address that ps->pi is already pointing to.
b. Since the address of the allocation didn't change, not updating ps->pi isn't a problem.
c. Updating ps->pi with the same address it already holds does no harm.
d. Catching the unchanged address in a different pointer, checking whether it's NULL (it isn't, in this case), then updating ps->pi with the same address it already holds does no harm.

When realloc expands but has to copy the existing allocation's data to a new location and free the existing allocation, the realloc call returns the new location's address.
b. This code doesn't store the returned updated location, so ps->pi is a dangling pointer to the old location, which is then dereferenced.
c. This code updates ps->pi with the new location, so all future access to ps->pi's elements uses their new addresses.
d. Catching the new location in a different pointer, checking whether it's NULL (it isn't, in this case), then updating ps->pi with the new address will allow all future access to ps->pi's element to use their new addresses.

When realloc fails to expand, it leaves the existing allocation in place unchanged and returns NULL. The next line after the comment in the function will attempt to access the non-existent expanded element of the allocation (recall that, since C arrays are indexed starting at 0, array[array_length] is out of bounds). Thus, in order to avoid accessing the array out of bounds, the code we add must avert this.
b. This code doesn't check the return value of realloc, so it has no way to know to avoid the subsequent OOB store.
c. This code updates ps->pi with the NULL return value. This loses the pointer to the unexpanded array (memory leak), and also results a segfault when the subsequent store dereferences this now-NULL pointer.
d. This code does check the return value of realloc, so ps->pi is correctly not updated -- so far, so good. With the check it also **could avoid the subsequent store beyond the end of the unexpanded array by adding something like "else return;" to the if statement ... but it doesn't, in fact, do that, and thus will still access the unexpanded array out of bounds.**

This exam's theme was YouTuber Sam Denby, with references (in italics) to his channels (Wendover Productions, Half as Interesting, Jet Lag: The Game, Extremities), collaborators, other ventures, and common content themes.

Q0: "The Logistics of X" is Sam's series of Nebula original videos
Q1: "Half As Interesting" is Sam's second YouTube channel; "Amy" references Amy Muller, an HAI writer also featuring in Sam's other works.
Q2: Wendover is his channel / production company name shoehorned into a terrible pun for "went over"; trains are an evergreen topic for Wendover and HAI, and the most common mode of transporation on JL:TG
Q3: Jet Lag: The Game is Sam's current fastest growing channel and the exam author's guilty pleasure. "Denby" is Sam's surname; "The Boys" is a common reference to HAI writers / JLTG game designers and competitors Ben Doyle and Adam Chase; "Hendy", "McManus", and "M.Khare" reference YouTuber/Nebula creator/JLTG contestants Toby Hendy (Tibees), Brian McManus (Real Engineering), and Michelle Khare (Challenge Accepted).
Q4: "Crime Spree" was an HAI miniseries that was a sort of trial run of JLTG concepts. At the time this exam was given, Season 9 of JLTG released on YouTube on the day this exam was given. (Season 10 was filmed the following week.)
Page 8 text: "mandatory rest period" is a JLTG game safety rule; "Extremities" is another of Sam's YouTube channels. "Nebula" is the creator-owned streaming service for which Sam is the chief content officer. "local snack" refers to the JLTG "Snack Zone" recurring bit.