Midterm Exam COS 217 Fall 2025

This exam consists of four 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 pages at the end 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: _____ NetID: _____

Precept:

- ◯ P01 - MW 1:20 Christopher Moretti
- ◯ P02 - MW 3:30 Amelia Dobis
- ◯ P03 - TTh 12:15 Kevin Alarcón Negy
- ◯ P04 - TTh 12:15 Andrew Johnson
- ◯ P05 - TTh 12:15 Nicholas Yap
- ◯ P07 - TTh 1:20 Lana Glisic
- ◯ P08 TTh 3:30 Kevin Alarcón Negy

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 Stats:
Total points: 65
Maximum: 60
Mean: 43.22
Median: 44
Standard Deviation: 8.8
Percentiles: 10th - 32 / 25th - 37 / 75th - 50 / 90th - 54

X_____

Question 1: Building and Running Programs 15 points

a. For each of the following errors or warnings seen while building or running a program, identify when the error/warning appears: list the stage of the build process or “runtime” if it is a runtime error. Write your answers in the boxes provided for each part.

(i)

```
Segmentation fault (core dumped)
```

Runtime

(ii)

```
/usr/bin/ld: client.c:(.text+0x20): undefined reference to `printf'
collect2: error: ld returned 1 exit status
```

Linker

(iii)

```
test.c:6:1: error: unterminated comment
```

Preprocessor

(iv)

```
test.c:12:14: warning: format '%d' expects argument of type 'int', but
argument 2 has type 'int *' [-Wformat=]
   12 |     printf("%d\n", ptr);
      |            ~^    ~~~
      |            |     |
      |            int   int *
      |            %ls
```

Compiler

b. The process of building a program utilizes the principle of modularity to increase efficiency. State in two sentences or less how it does this and how the increase in efficiency is achieved.

A strong answer should have mentioned at least one of the following,
and had to have been in the context of aiding in the efficiency of building a program:
a) Allowing for partial builds (reducing unnecessarily repeated computation)
b) Allowing for parallelization of compilation (reducing overall build time)
c) Reducing debugging space to specific stages (reducing human intervention effort)

c. Complete the table below with the four stages of the process of building an executable starting from C source code. Write the stages in order (i.e., the first stage to be executed will be in row 1., the last stage will be in row 4.). For each build stage, indicate:

- the name of the stage
- the format of the file(s) produced by that stage
  (Answer with a letter from the options given – you will *not* use every option.)
- whether or not that format is considered human-readable (Answer YES or NO.)

File Format Options:

A: Assembly language **B**: Bytecode **C**: C source code **D**: DFA
E: Executable file **F**: Machine language **G**: Makefile

	Stage Name	File Format	Human Readable
1.	Preprocessor	C	YES
2.	Compiler	A	YES
3.	Assembler	F	NO
4.	Linker	E	NO

Question 2: Modularity

14 points

A COS217 student wrote a program with two functions that each double every element of an integer array – one version using a `for` loop and arithmetic operations, the other using a `while` loop with pointers and bitshifting. The `main` function uses a helper function that prints an array to show the results of each version.

Filename: **`double.c`**

```
01  #include <stddef.h>
02  #include <stdio.h>
03  #include <assert.h>
04  enum {ARRAY_LENGTH = 5};
05
06  void double_array_elements_for_loop(int aiNums[], size_t n);
07  void double_array_elements_while_loop(int aiNums[], size_t n);
08  void print_array(int arr[], size_t n);
09
10  int main(void) {
11     int aiNums1[ARRAY_LENGTH] = {4, 1233, 8, -32, 0};
12     int aiNums2[ARRAY_LENGTH] = {4, 1233, 8, -32, 0};
13
14     double_array_elements_for_loop(aiNums1, ARRAY_LENGTH);
15     print_array(aiNums1, ARRAY_LENGTH);
16
17     double_array_elements_while_loop(aiNums2, ARRAY_LENGTH);
18     print_array(aiNums2, ARRAY_LENGTH);
19     return 0;
20  }
21
22  void double_array_elements_for_loop(int aiNums[], size_t n) {
23     /* implementation correct but not shown */
24  }
25
26  void double_array_elements_while_loop(int aiNums[], size_t n) {
27     /* implementation correct but not shown */
28  }
29
30  void print_array(int arr[], size_t n) {
31     /* implementation correct but not shown */
32  }
```

Modularize this program. The result should be that you can build and run two versions – one that uses the `for` loop implementation and one that uses the `while` loop implementation – **using the same `main` function and print function for each**. The revised client should call an array doubling function only once, instead of running both implementations sequentially like the original `main` function. Your module should use the best practices from the last version of `IntMath` from precept and `Str` from Assignment 2.

As much as possible, reuse the given code from `double.c` in your new files by writing down specific line numbers in each box below (e.g., "9" or "10-12"). If you want to refer to a line but make a change to it, write, e.g., "17, but add/remove/change …". If you want to write a new line of code, do so in the appropriate box in its proper place among the reused and modified lines.

You might not have to use every line of code from the file `double.c`.

Label the filenames and write the line numbers you want to include in each file (in order) in the boxes below, one file per box.

New filename: double.h

```
#ifndef DOUBLE_INCLUDED
#define DOUBLE_INCLUDED
01
/* function comment for 06 */
06 renamed just double_array_elements
#endif
```

New filename: doublefor.c

```
#include "double.h"
03
22 renamed just double_array_elements
23-24
```

New filename: doublewhile.c

```
#include "double.h"
03
26 renamed just double_array_elements
27-28
```

New filename: testdouble.c

```
#include "double.h"
02
03 (optional to assert in static fun)
04
08 (optional if 30-32 above 10)
10-11
14 renamed just double_array_elements
15
19-20
/* function comment for 30 */
30 with added static keyword
31-32
```

Question 3: Indirection and Portability

8 points

Consider the following C program. Assume the program is built and run on armlab.

```
#include <stdio.h>
enum {ARRAY_LENGTH = /* redacted */ };

void print_array(char ac[]) {
   char c;
   int *pi = (int*) &ac[0];

   while((c = *(char*)pi) != '\0') {
      putchar(c);
      pi++;
   }
}

void fill_array(char ac[]) {
   size_t i;
   for(i = 0; i < ARRAY_LENGTH - 1; i++)
      ac[i] = 'A' + i;
   ac[ARRAY_LENGTH - 1] = '\0';
}

int main(void) {
   char ac[ARRAY_LENGTH];
   fill_array(ac);
   print_array(ac);
   return 0;
}
```

a. Assume `ARRAY_LENGTH` is defined as `5`. What are the contents of `ac` in main after returning from `fill_array`?

A	**B**	**C**	**D**	**\0**

b. Assume `ARRAY_LENGTH` is defined as `13`. What is the output printed in `print_array`?

AEI because pi++ increments by 4 bytes

c. Setting `ARRAY_LENGTH` to `20` leads to undefined behavior on armlab. Explain in 1 sentence why this is the case.

When ARRAY_LENGTH is 20, the trailing nullbyte from fill_array will be at index 19, so as pi++ increments by 4 from 16 to 20, it will skip over the element that was supposed to trigger the loop's end condition and result in reading off out of bounds off the end of the array, triggering indeterminate behavior.

d. Explain in 1 sentence why the answers to parts **b.** and **c.** above might not be correct on all systems. Specifically, why does whether `ARRAY_LENGTH` is defined as `13` or `20` result in defined or undefined behavior depending on a system-dependent characteristic of C?

sizeof(int) (which is how many bytes pi++ advances) is not precisely specified by the C standard, and thus may be of different sizes on different machines (so those machines would have a different set of viable ARRAY_LENGTH values to avoid reading off the end of the array)

Question 4: Bugs and the Stack 28 points

Consider the following implementation of `array_equals` that checks if the content in two arrays is equal. Assume that all needed header files have been included.

```
01 int array_equals(const int* arr0_p, const int* arr1_p, size_t size) {
02    int* arr_elem = &arr1_p[0];
03    size_t i;
04    for(i = 0; i < size; i++) {
05       if(arr0_p[i] != *(arr_elem++))
06          return 0;
07    }
08    return 1;
09 }
```

a. One of the lines in `array_equals` does not compile cleanly. Write which line number it is, and provide a corrected version of the line.

Line 2 assigns a const int* pointer to a (non-const) int *. This will be a compiler warning.
Fix: casting away const or changing variable type, e.g., const int* arr_elem = &arr1_p[0];

Also consider this function that calls `array_equals`.

```
int main() {
   int arr0[4] = { 2, 3, 4, 2 };
   int arr1[3] = { 2, 3, 4 };
   int eq = array_equals(arr0, arr1, 4);
   printf("%d", eq);
   return EXIT_SUCCESS;
}
```

b. Complete the contents of the table showing the contents of the stack section of memory **just before we return from the `array_equals` function**.

For simplicity, we assume that the last entry for the main stack frame is at **line 2000**. Addresses in our listing grow towards the bottom of the table, i.e., the rows *above 2000* should all contain **smaller** addresses than 2000. The addresses should be consistent with the size of each variable's type on armlab.

Assume that parameters and variables are pushed onto their function's stack frame in the order in which they are declared, and there are no gaps or padding used.

The table contains **exactly** the number of rows you need to fill in or complete. Some entries are already provided to help you get started.

Write the array index into the two partially filled in cells in the bottom two rows and fill in the rest of the table, except for the greyed out cell.

Address	Variable Name	Contents	Function Name
1932	`i`	4	`array_equals`
1940	`arr_elem`	1992	`array_equals`
1948	`size`	4	`array_equals`
1956	`arr1_p`	1976	`array_equals`
1964	`arr0_p`	1988	`array_equals`
1972	`eq`		`main`
1976	`arr1[0]`	2	`main`
1980	`arr1[1]`	3	`main`
1984	`arr1[2]`	4	`main`
1988	`arr0[0]`	2	`main`
1992	`arr0[1]`	3	`main`
1996	`arr0[2 ]` (fill in index)	4	`main`
2000	`arr0[3 ]` (fill in index)	2	`main`

c. This program outputs `1`, which is incorrect: the arrays do **not** have the same elements! In one sentence, what is the design flaw in the program that causes the incorrect output?

array_equals assumes both arrays are the same size, and should have instead taken separate size parameters for the two arrays

(Question 4 was the last question on this midterm exam. This page can be used as scratch space. Please note that anything written on this page will not be graded.)