COA1 Exam 3 – Fall 2018

Name: ____________________________ Computing ID: __________

Write Letters clearly: if we are unsure of what you wrote you will get a zero on that problem.
Bubble and Pledge the exam or you will lose points.
Assume unless otherwise specified:

- all necessary `#include`s have been used
- `char`, `short`, `int`, and `long` are 8-, 16-, 32-, and 64-bits long, respectively
- compilation happens using `clang` on a Linux system

Single-select by default: Multiple select are all clearly marked; answer them by putting 1 or more letters in the box, or writing "`none`" if none should be selected.
Mark clarifications: If you need to clarify an answer, do so, and also add a * to the top right corner of your answer box.

....

Information for questions 1–3
For each of the following, identify the bug in the code and how to fix it. Assume that all used variables have been defined.

Question 1 [2 pt]: (see above)

```
int *array = malloc(n);
for (int i=0; i<n; i+=1) array[i] = i*i;
```

____

____

Question 2 [2 pt]: (see above)

```
char word[64];
scanf("%s", word);
```

____

____

Question 3 [2 pt]: (see above)

```
foo *x = calloc(1, sizeof(foo));
x.y = 3;
```

____

____

Information for questions 4–5
The following code exercises undefined behavior

```
int main(int argc, const char *argv[]) {
    char *all;
    int space = 0;
    int got = 0;
    for(int i=1; i<argc; i+=1) {
        const char *arg = argv[i];
        while (*arg) {
            if (got == space) {
                space = (space + 4)*2;
                all = realloc(all, space);
            }
            all[got] = *arg;
            got += 1;
            arg += 1;
        }
    }
    puts(all);
}
```

Question 4 [2 pt]: (see above) What is the undefined behavior and what problem does it cause if it does not behave the way the program writer intended?

Question 5 [2 pt]: (see above) What does this program do if the undefined behavior does not cause a problem? Include both an example and a description, such as might go in the **description** and **examples** sections of a manual page.

Question 6 [2 pt]: Write an implementation of `getline` using only `read` and the following implmentation of a `FILE`:

```
typedef struct {
    int fd; // the file descriptor of an open file
    char buffer[128]; // for buffered input
    int i, len, flags; // to be used, or not, as you please
} FILE;
```

Your implementation must read a full buffer at a time if there is sufficient data, must work correctly if lines are larger than the buffer or multiple lines fit in a single buffer. You must meet the specification given in the manual page excerpt at the end of this exams.

You may assume that on its first call, `getline` gets a valid `fd` of a newly-opened file and the rest of its `FILE` argument's bytes are set to 0. Subsequent calls for the same `FILE` should retrieve subsequent lines.

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream) {
```

Question 7 [2 pt]: Garbage collectors are able to reclaim

A all garbage on the heap and stack
B all garbage on the heap
C all garbage on the stack
D all unreachable memory on the heap and stack
E all unreachable memory on the heap
F all unreachable memory on the stack

Answer:

Question 8 [2 pt]: If a function returns a `char *` and its manual page does not specify where in memory the data it points to lives or anything else about its memory, it is likely to be
Select all that are likely by putting 0 or more letters in the box.

A a pointer into data you passed it
B a pointer to heap-allocated memory
C a pointer to global memory
D a pointer to stack memory

Answer:

Information for questions 9–10
Consider the following program:

```
int main() {
    char data[8];
    ssize_t got = read(0, data, 8);
}
```

Question 9 [2 pt]: (see above) If you run this program and immediately type Ctrl+D, what value will be in the variable `got`? Answer as a decimal integer, like `23`.

Answer:

Question 10 [2 pt]: (see above) What are the `char` values in `data` if you run this program, typing `23` then the enter key, then Ctrl+D? Write a "?" if the value of some byte is undefined.

Index:	0	1	2	3	4	5	6	7
Char:								

Pledge:

On my honor as a student, I have neither given nor received aid on this exam.

Your signature here

NAME — malloc, free, calloc, realloc

SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

DESCRIPTION

The **malloc**() function allocates *size* bytes and returns a pointer to the allocated memory. *The memory is not initialized.* If *size* is 0, then **malloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

The **free**() function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc**(), **calloc**(), or **realloc**(). Otherwise, or if *free(ptr)* has already been called before, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

The **calloc**() function allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero. If *nmemb* or *size* is 0, then **calloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

The **realloc**() function changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will *not* be initialized. If *ptr* is NULL, then the call is equivalent to *malloc(size)*, for all values of *size*; if *size* is equal to zero, and *ptr* is not NULL, then the call is equivalent to *free(ptr)*. Unless *ptr* is NULL, it must have been returned by an earlier call to **malloc**(), **calloc**(), or **realloc**(). If the area pointed to was moved, a *free(ptr)* is done.

RETURN VALUE

The **malloc**() and **calloc**() functions return a pointer to the allocated memory, which is suitably aligned for any built-in type. On error, these functions return NULL. NULL may also be returned by a successful call to **malloc**() with a *size* of zero, or by a successful call to **calloc**() with *nmemb* or *size* equal to zero.

The **free**() function returns no value.

The **realloc**() function returns a pointer to the newly allocated memory, which is suitably aligned for any built-in type and may be different from *ptr*, or NULL if the request fails. If *size* was equal to 0, either NULL or a pointer suitable to be passed to **free**() is returned. If **realloc**() fails, the original block is left untouched; it is not freed or moved.

NAME — read

SYNOPSIS

```
#include <unistd.h>
ssize_t read(int fildes, void *buf, size_t nbyte);
```

DESCRIPTION

The *read*() function shall attempt to read *nbyte* bytes from the file associated with the open file descriptor, *fildes*, into the buffer pointed to by *buf*.

On a regular file, the *read*() shall start at a position in the file given by the file offset associated with *fildes*. The file offset shall be incremented by the number of bytes actually read.

No data transfer shall occur past the current end-of-file. If the starting position is at or after the end-of-file, 0 shall be returned. If the file refers to a device special file, the result of subsequent *read*() requests is implementation-defined.

RETURN VALUE

Upon successful completion, this function shall return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions shall return -1 and set *errno* to indicate the error.

NAME — getline

SYNOPSIS

```
#include <stdio.h>

ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

DESCRIPTION

getline() reads an entire line from *stream*, storing the address of the buffer containing the text into **lineptr*. The buffer is null-terminated and includes the newline character, if one was found.

If **lineptr* is set to NULL and **n* is set 0 before the call, then **getline**() will allocate a buffer for storing the line. This buffer should be freed by the user program even if **getline**() failed.

Alternatively, before calling **getline**(), **lineptr* can contain a pointer to a **malloc**(3)-allocated buffer **n* bytes in size. If the buffer is not large enough to hold the line, **getline**() resizes it with **realloc**(3), updating **lineptr* and **n* as necessary.

In either case, on a successful call, **lineptr* and **n* will be updated to reflect the buffer address and allocated size respectively.

RETURN VALUE

On success, **getline**() returns the number of characters read, including the delimiter character, but not including the terminating null byte ('\0'). This value can be used to handle embedded null bytes in the line read.

This function returns -1 on failure to read a line (including end-of-file condition). In the event of an error, *errno* is set to indicate the cause.