CSO1 **Name:** _____
Spring 2022
Midterm 2
2022-04-24
Time Limit: 75 minutes Computing ID _____

Instructions:

1. This exam contains 8 pages (including this cover page) and 13 questions.
2. You have **75 minutes** to complete the examination.
3. Write your answers in this booklet. We scan this into GradeScope, so **please try to avoid writing on the backs of pages**.
4. If a question presents several options in a list, mark the bubble next to the one correct answer. All such questions on this test are single-select.
5. You may not use a calculator or notes.
6. Because this assessment is being given in several places, we cannot fairly answer questions during it. If you find a question ambiguous or unclear, please explain that on the page by the question itself and we will consider your explanation during grading.
7. We will use the following data type sizes:

Types	size in bits
`char`	8
`short`	16
`int` and `float`	32
`long` and `double`	64

8. The next page contains a copy of the manual page for `malloc`, `free`, `calloc`, and `realloc`, which you are welcome to refer to during the test if you would like.
9. Please sign the below Honor Code statement.

I have neither given nor received aid on this exam.

Signature: _____

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.

1. For each of the following questions, assume the first eight registers have the following values prior to the assembly being run:

```
rax = 0x0000000000000001
rcx = 0xffffffffffffffff
rdx = 0x0000000000000010
rbx = 0x00000000000000ff
rsp = 0x0000000000000010
rbp = 0x0000000000000030
rsi = 0x00000000ceacbdef
rdi = 0x0000000010010100
```

Note: the questions are independent. Do not use the result of one as the input for the next. Answer by writing a changed register and its new value, like "`RDI = 24F2`", leaving one or more lines blank if fewer registers change than there are lines.

Please write **the full 64-bit register (in hex)** even if the instruction only modifies part of it.

(a) Which program registers are modified, and to what values, by `leaq 0x10(%rcx,%rdx,4), %rax`?

(b) Which program registers are modified, and to what values, by `pushq %rcx`?

(c) Which program registers are modified, and to what values, by `retq`?

(d) Which program registers are modified, and to what values, by `addq %rsi, %rdi`?

(e) Which program registers are modified, and to what values, by `movl %ecx, %edx`?

2. Consider the following program.

```
int main() {
    char  a[3] = {'p', 'q', 'r'};
    char  b[3] = {'s', 't', 'u'};
    char  c[3] = {'w', 'x', 'z'};
    char *d[3] = {a, b, c};
    char *e = d[1];
    char *f = e + 2;
    char g = *f;
    printf("%c\n", g);
    return 0;
}
```

Note: `printf("%c\n", g);` means "print the character stored in variable `g`", so if `g='a'` it would print `a`.

What get prints out? If the program would crash write `crash`.

3. Which of the following represents the most correct use of malloc?

- ◯ `int x = malloc(sizeof(int));`
- ◯ `int *x = (void *)malloc(sizeof(int*));`
- ◯ `int *x = (int *)malloc(sizeof(int));`
- ◯ `int *x = (int *)malloc(sizeof(int*));`

4. Consider the following code snippet. What is value of the variable `c`?

```
unsigned long a = 512;
char b = (char) a;
int c = (int) b;
```

5. Consider the following code snippet.

```
short a = -1;
long b = (long) a;
```

What is the value of `b`?

- ◯ `0x0000000000000000`
- ◯ `0x000000000000FFFF`
- ◯ `0xFFFFFFFFFFFF0001`
- ◯ `0xFFFFFFFFFFFFFFFF`

6. Consider this C code:

```
#define IS_ODD(num)   ((num) & 1)
int x = IS_ODD(y);
```

This code will compile to:

- ◯ a `call` instruction to the function `IS_ODD`
- ◯ a `testl` instruction
- ◯ an `andl` instruction
- ◯ multiple `call` instructions
- ◯ nothing at all

7. If a function parameter is larger than 8 bytes, where is it stored?

- ◯ the stack
- ◯ global memory
- ◯ the heap
- ◯ the code section of memory

8. Where is the memory allocated using `malloc` or `calloc` is stored?

- ◯ the stack
- ◯ global memory
- ◯ the heap
- ◯ the code section of memory

9. Where are immediate values stored?

- ◯ the stack
- ◯ global memory
- ◯ the heap
- ◯ the code section of memory

10. Consider the following code snippet?

```
int a[2] = {1, 2};
int c = a[0] + a[1];
return c;
```

Rearrange the following assembly instruction so that it implements the code above.

1. `addl 8(%rsp), %rax`
2. `movl 0(%rsp), %rax`
3. `pushq $1`
4. `pushq $2`
5. `ret`

Write the numbers of the instructions in the boxes below in the correct order.

11. The following program contains at least one memory leak. Fill in any locations where you need to free the pointer(s).

```
int main(){
  char *str;


  str = (char *) calloc(15, sizeOf(char));


  str = (char *) realloc(str, sizeOf(char)* 25);


  return 0;
}
```

12. Given the following struct

```
typedef struct {
  float score;
  char grade;
} Student;
```

Implement a function that combines two arrays of the same length (an array of scores and an array of grades) into an array of `Student` stucts. For example, given `score = {1.7, 100.0, 97.0}` and `grade = {'F', 'A', 'CR'}` return the array of student structs `{{1.7, 'F'}, {100.0, 'A'}, {97.0, 'CR'}}`.

Do not check that numeric scores and letter grade match, simply pair them up.

We expect a correct solution to use `malloc` and/or `calloc`, but not to use `free`.

```
#include <stdlib.h> // contains malloc, calloc, free, and realloc
Student *combine(float *scores, char *grades, int number_of_students) {
```

}

13. How difficult was this midterm?

- ◯ too easy
- ◯ easy but fair
- ◯ fair
- ◯ difficult but fair
- ◯ too difficult