# CS 107 Midterm, Winter 2022, Student: cgregg

# Instructions

There are **four** problems on this exam. Here are some details:

- All the problems involve writing one or more functions.
- You will not be able to run your code, but when you submit, you will be shown if there are compile errors.
- When you click any of the "Submit" buttons, all of your code for all problems will be submitted.
  - You should get a pop-up message confirming that all of your code has been submitted.
- You have **120** minutes to complete the exam (plus two minutes to read these instructions). When the timer runs out, you will not be able to submit again, and you will not be able to see your final answers.
- We will grade your final submission, and you can submit multiple times if you wish. We suggest submitting your code after each problem you solve.

# Problem 1: Bits and Bytes (10 points)

(Exam time left: 121:27)

Write the function:

```
unsigned int bits_range(unsigned int num, int left_bit, int right_bit);
```

This function returns an `unsigned int` that is comprised only of the bits between `left_bit` and `right_bit`, inclusive, in `num`, shifted to the right so that `right_bit` is now at bit position 0. Consider the number `0xff00ff`. Its bits are as follows (bit indexes on the top row):

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
-----------------------------------------------------------------------------------------------
 0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1
```

A call to `bits_range(0xff00ff, 16, 3)` would return only the bits 16 to 3 (inclusive), and would shift the number to the right three bits. In other words, it would mask and shift the number so that only bits 16 to 3 are in the return value. Here are the bits in the original number that we want to return:

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
-----------------------------------------------------------------------------------------------
 0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1
                                                 ----------------------------------
```

And here is the return value, 8223 (0x201f):

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
-----------------------------------------------------------------------------------------------
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  1  1  1  1  1
                                                       ----------------------------------------
```

Click here to go to a test page where you can test other examples.

We have given you starter code below that includes the potentially-helpful `ALL_ONES` that is a number with all bits set to 1.

Restrictions:

- `left_bit` will always be greater than or equal to `right_bit`. Both numbers will be less than 32.
- You are not allowed to have any loops in your function.
- You are not allowed to left-shift bits from the original number in order to "shift them out to the left."
- You are not allowed any conditional statements.

(Exam time left: 121:27)

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

unsigned int bits_range(unsigned int num, int left_bit, int right_bit) {
    // bits are numbered from right to left:
    // bit 0 is the least significant, and bit 31 is the most significant
    // left_bit is the left-most bit in the number to include
    // right_bit is the right-most bit in the number to include

    unsigned int ALL_ONES = ~0;

    return 0;
}

```

Submit

# Problem 2: Strings, pointers, and copying memory (15 points)

(Exam time left: 121:27)

Write the function `bracket_string`, which takes a string, a substring, and a "bracket" string. The function finds the first occurence of the substring and adds the bracket string on both sides of the substring. Here is an example:

Original string: `hello my name is dog`
Substring: `name`
Bracket: `***`

Result: `hello my ***name*** is dog`

Here is another example:

Original string: `The number 42 is special.`
Substring: `42`
Bracket: `FT`

Result: `The number FT42FT is special.`

The function modifies the *original* string, which is guaranteed to have enough space to hold the original string plus two copies of the bracket string, plus a null-terminator.

Restrictions:

- For full points, your code should work entirely on the original string *in-place*. You may create as many pointers or integers as you would like, but you will not get full points if you create any temporary buffer space to hold strings (but, if that is the only way you can think of solving the problem, you can still receive up to 10 points for your solution).
- Note that `strcpy` *does not work on overlapping strings*. In other words, if you want to copy one part of a string into an overlapping memory location, you must use `memmove` instead of `strcpy`.
- Recall that `strncpy` *does not null-terminate after copying if there are no null bytes in the number of bytes requested.*
- If the substring does not occur in the original string, you should not make any changes to the original string.

(Exam time left: 121:27)

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

/* bracket_string finds the first occurrence of substr in s
 * and puts the characters in bracket on both sides of substr
 * in s. s is guaranteed to have enough space to hold the original
 * string plus two copies of bracket. If substr is not found in
 * s, s is left unchanged.
 */
void bracket_string(char *s, const char *substr, const char *bracket) {

}
```

Submit

# Problem 3: Dynamic memory management (10 points)

For this problem, write the function `print_longest_line()` which takes in a file pointer and uses your (presumably working) `read_line()` function

from assignment 3 to read all the lines in a file and print out the longest line. If there are more than one longest line, your function should **print** the line closest to the end of the file (*not return it*). Your function should only store the current longest line as you read through file, and remember that your function is responsible for freeing any pointers you receive from `read_line`. Also remember that `read_line` returns `NULL` when it reaches the end of the file it is reading.

(Exam time left: 121:27)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  char *read_line(FILE *file_pointer);
6
7  /* print out the longest line in a file, keeping track of only
8   * a single line at a time. If there are no lines in the file
9   * don't print anything.
10  */
11 void print_longest_line(FILE *file_pointer) {
12
13 }
14
```

Submit

# Problem 4: Generics and function pointers (15 points)

Write a generic function, `generic_sum` that has the ability to sum the values of any type passed into it. Because the function is generic, it does not know any details about what a "sum" is, so it needs to rely on a function pointer for a function that can add a number to a total for whatever type the array holds.

This is the function signature for `generic_sum`:

```
void generic_sum(void *arr, int nelems, int width,
                 void *total, void (*add)(const void *, void *));
```

The `add` function takes two parameters: the first is a pointer to one element in the array, and the second is the `total` pointer.

Consider an array of `ints`. The `generic_sum` function walks through the array and passes in each element pointer, as well as the `total` pointer, to the `add` function. An `add` function simply adds the

element to the total and returns. One nice feature of an `add` function is that the type of the array elements does not have to have the same width as the type for the total. So, for example, an array of `chars` could be added together with the sum going into an `int`. The `generic_sum` function need not concern itself with that detail, but an individual `add` function necessarily needs to know that information.

For this problem, write the `generic_sum` function and two `add` functions, one that will sum an array of `chars` and sum the maximum individual values of an array of `point structs`. See the code below for the declaration of a `point struct`, also with the sum going into an `int`.

We have provided you with an example `main` function for this problem, to see how the generic function behaves. (Exam time left: 121:27)

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

typedef struct point {
    int x;
    int y;
} point;

void generic_sum(void *arr, int nelems, int width,
                 void *total, void (*add)(const void *, void *)) {

}

void add_char(const void *element, void *total) {

}

void add_point_max(const void *element, void *total) {

}

// DO NOT CHANGE CODE BELOW THIS LINE
int main(int argc, char **argv)
{
    // sum the ascii values of the characters in a string
    int total = 0;
    char *s = "hello"; // ascii values: h: 104, e: 101, l: 108, o: 111
    // sum of "hello": 104 + 101 + 108 + 108 + 111 = 532
    generic_sum(s, strlen(s), 1, &total, add_char);
    printf("total: %d\n", total);

    // sum the maximum values of an array of points
    point points[] = {{1, 5}, {3, 2}, {0, 1}, {4, 4}};
    // values to add:     ^    ^          ^    ^
    // 5 + 3 + 1 + 4 = 13
    total = 0;
    generic_sum(points, sizeof(points)/sizeof(points[0]), sizeof(point),
                &total, add_point_max);
    printf("total: %d\n", total);

    return 0;
}
```

Submit

code font size % default