IMPORTANT NOTES for all questions on the exam: You may need to scroll vertically or horizontally to fully view blocks of code. Solutions that violate any specified restrictions may get partial credit. For these problems, you do not need to worry about calling `assert` to check for heap errors. There is 1 point allocated on the exam per minute of the original 1hr 50min time.

1) Sticky Search 20 Points/110 Total

Fill in the blanks below to implement a function `findMatch` that takes in a string `searchIn` to search, a string `searchFor` to search for, and whether the search should stick to the front or back, and returns a pointer to the first occurrence of `searchFor` in `searchIn` that fits the specified restrictions, or NULL if no valid match was found.

```c
char *findMatch(char *searchIn, char *searchFor, bool stickToFront, bool stickToBack);
```

If the search is not stuck to the front or to the back, then it's a regular string search - in other words, return the first occurrence of `searchFor` anywhere in `searchIn`. If the search is "stuck" to the front, a valid match must be at the start of `searchIn`. If the search is "stuck" to the back, a valid match must be at the end of `searchIn`. If the search is "stuck" to both the front and back, a valid match must be simultaneously at the front and the back of `searchIn` (in other words, it must match `searchIn` exactly). Here are some search examples - let's say `searchFor` is always "cs107":

- If `stickToFront` and `stickToBack` are both false, then any occurrence of "cs107" in the string is a valid match, and we should return a pointer to the first one. E.g. if `searchIn` is "woocs107woo", the function would return a pointer to the "c" in "cs107".
- If `stickToFront` is true and `stickToBack` is false, then only an occurrence of "cs107" at the start of the string is a valid match. E.g. if `searchIn` is "cs107woo", the function would return a pointer to the "c" in "cs107", but if it were "woocs107woo" it would return NULL.
- If `stickToFront` is false and `stickToBack` is true, then only an occurrence of "cs107" at the end of the string is a valid match. E.g. if `searchIn` is "woocs107", the function would return a pointer to the "c" in "cs107", but if it were "cs107woo" it would return NULL.
- If `stickToFront` is true and `stickToBack` is true, then only an occurrence of "cs107" that is both at the start and end of the string is a valid match. E.g. if `searchIn` is "cs107", the function would return a pointer to the "c" in "cs107", but if it were "cs107woo",
Fill in each blank to complete the implementation of `findMatch`. You may assume that `searchIn` and `searchFor` are valid C strings that are not NULL, but they may be empty strings. You should use built-in string library functions whenever possible. **Blanks 1, 2 and 4 must be valid C expressions. Blank 3 may be 0 or more lines of C code long.**

```c
char *findMatch(char *searchIn, char *searchFor,
                 bool stickToFront, bool stickToBack) {
    if (stickToFront && stickToBack) {
        if (___1___) {
            return searchIn;
        }
    } else if (stickToFront) {
        if (___2___) {
            return searchIn;
        }
    } else if (stickToBack) {
        ___3___
    } else {
        return ___4___;
    }
    return NULL;
}
```
2) Spring Cleaning  25 Points/110 Total
A) Complete the function `springCleaning` below. This function takes in an array of pointers to heap-allocated integers, which were allocated on the heap separately, so they may be spread out all over the heap. The array itself may be allocated on either the stack or the heap - it's only the integers being pointed to that are guaranteed to be heap-allocated. The function also takes in the number of pointers in the array. Your function should update this array by going through one pair of pointers at a time and updating both pointers to point to the same heap-allocated integer that is the sum of the values they previously pointed to. Here's a before-and-after diagram to visualize this:

Before:

```
1
15
2
-4
100
7
```

After:

```
16
-2
107
```

Complete the implementation of `springCleaning`. You may assume that neither the array itself nor any of its contained pointers are `NULL`, and that the array has an even number of elements. Your implementation **should not allocate any new heap memory**; instead, you should re-use the existing heap memory to store the new values. Any heap memory that is no longer referred to by the array should **be freed** at the appropriate time.
B) One of your teammates wants to write a program that uses your `springCleaning` function. Over the course of their program, they allocate an array of pointers to heap-allocated integers, pass that array to your function, and then print out the elements in the resulting array before freeing them. They have provided you an excerpt of the code with some comments - the ellipses (...) indicate code that was omitted. There are two core memory issues in the provided code; briefly identify each issue and how to correct it, using no more than 3 sentences per issue. (You don't need to write the code to fix it, just explicitly describing the issue and the fix is sufficient).

```c
// Make a heap array of pointers to heap-allocated integers
int nelems = 6;
int **ptrs = malloc(nelems);
assert(ptrs != NULL);

// assume array is initialized to look like the "before" diagram ...

springCleaning(ptrs, nelems);

/* assume array now looks like the "after" diagram *
 * with only that memory still allocated */
...

// Print out and free all the consolidated elements
for (int i = 0; i < nelems; i++) {
    printf("%d\n", *(ptrs[i]));
    free(ptrs[i]);
}
free(ptrs);
```
Implement the function `getDiceRolls` that returns a heap-allocated array of random "dice rolls" (integers from 1 to 6) of a random length. To implement this function, you should repeatedly call the provided `rollDice` function (assume this is implemented for you) which takes no parameters and returns a new random integer between 1 and 6. Each time you get a dice roll, you should add that to the next position in your array while that dice roll value is not equal to 1. As soon as you roll a 1, you should stop, not include that roll in the array, and return the array. The rolls should be stored in the array in the same order they were generated from the `rollDice` function. As an example, let's say we first roll a 2; we would add that to the first slot in our array. Then we roll again and get a 3; we would add that to the second slot in our array. Then we roll a 1; we stop and return the array [2, 3].

Since you do not know initially how big the resulting array will be, you should use a resize-as-you-go approach. The array should start out with space for two elements. Every time you need more space, the new array should be enlarged to add space for two additional elements. You should only enlarge the array when you know that you need more space. It is the caller's responsibility to free the array when no longer needed.

One additional detail about `getDiceRolls`: it also takes one parameter, which is where the length of the new array should be stored when the function finishes. In other words, after `getDiceRolls` finishes executing, the location pointed to by `lengthPtr` should contain the number of dice rolls in the array. This is needed so the caller knows how long the dice array is. You may assume that `lengthPtr` is not NULL.
/* This function returns a new random number between 1 and 6. 
 * Assume this function is already implemented for you. 
 */
int rollDice() { ... }

...

int *getDiceRolls(int *lengthPtr) {
    // your code here
}

...

// Example usage
int main(int argc, char *argv[]) {
    int length;
    int *rolls = getDiceRolls(&length);
    for (int i = 0; i < length; i++) {
        printf("%d
", rolls[i]);
    }

    free(rolls);
    return 0;
}
The generic function `containsLargerThan` returns whether or not an array contains an element larger than another specified element, as dictated by the provided comparison function.

```c
bool containsLargerThan(void *base, size_t nelems,
                        size_t elem_size_bytes, void *elem,
                        int (*cmp_fn)(const void *, const void *));
```

As an example, let's say we call `containsLargerThan` with the following integer array with 4 elements:

```c
[3, 1, 4, 6]
```

If we provided a comparison function that orders the numbers in ascending order, and we passed in the array and the separate element 5, the function would return `true` because according to the provided comparison function, there is an element that is "larger" than (comes after) 5. If we provided a comparison function that orders the numbers in descending order, and we passed in the array and the separate element 1, the function would return `false` because according to the provided comparison function, no element in the array is "larger" than (comes after) 1.

The function signature and parameters are specified as follows:

```c
bool containsLargerThan(void *base, size_t nelems,
                        size_t elem_size_bytes, void *elem,
                        int (*cmp_fn)(const void *, const void *));
```

- `base`: a pointer to the first element of an array
- `nelems`: the number of elements in the provided array.
- `elem_size_bytes`: the size of a single provided array element, in bytes.
- `elem`: a pointer to the element that we want to see if the array contains something larger than. In other words, if this is a pointer to the integer 6, then this function should return `true` if the array contains an element larger than 6 (according to the provided comparison function), or `false` otherwise. You can assume that `elem` points to the same type of element as is stored in the provided array.
- `cmp_fn`: a function pointer that accepts two parameters, both pointers to elements to compare, and returns a negative number if the first parameter is considered less than the second, 0 if the first parameter and the second parameter are considered equal, or a
positive number if the first parameter is considered larger than the second.

We assume that base, elem and cmp_fn are not NULL, but the array may be empty.
A) Write the containsLargerThan function.

```c
bool containsLargerThan(void *base, size_t nelems,
                  size_t elem_size_bytes, void *elem,
                  int (*cmp_fn)(const void *, const void *)) {
    for (int i = 0; i < nelems; i++) {
        // your code here
    }
    return false;
}
```
B) Implement the `cmpStringsDesc` function that could be used as a parameter to `containsLargerThan` with an array of strings to see if the array contains a string **shorter** than a specified string. In other words, this comparison function should order strings by length such that a **shorter** string is considered "larger" than a **longer** string. It should return a negative number if the first parameter is considered less than the second, 0 if the first parameter and the second parameter are considered equal, or a positive number if the first parameter is considered larger than the second.

```c
int cmpStringsDesc(const void *a, const void *b) {
    // your code here
}
```
For this problem, let there be a new variable type called `uchar` that is the exact same as an `unsigned char`, but just with a different name. It can be used exactly like an `unsigned char`, and the two are interchangeable.

Implement the following functions that manipulate `uchar`s.

You are not allowed to use any of the following operators in your answers for this problem: + (addition), - (subtraction), * (multiplication), / (division), % (modulus).
A) Write the function `filluchars` which takes in 4 `uchar`s and returns an unsigned integer with the `uchar`s placed into the int such that `a` is the highest-order byte, `b` is the second-highest-order byte, `c` is the third-highest-order byte, and `d` is the lowest order byte. For example: `filluchars(0x01, 0xab, 0xcd, 0xef)` would return `0x01abcdef`.

```c
unsigned int filluchars(uchar a, uchar b, uchar c, uchar d) {
    // your code here – remember, no prohibited math operators!
}
```
B) Write the function pulluchar which returns the uchar from the value at position ucharLocation, where ucharLocation = 0 is the lowerst order uchar, and ucharLocation = 3 is the highest order uchar. Here are some examples:

- pull_uchar(0x01abcdef, 0) would return 0xef
- pull_uchar(0x01abcdef, 3) would return 1.

```c
uchar pulluchar(unsigned int value, int ucharLocation) {
    // your code here – remember, no prohibited math operators!
}
```