

## ✔ CS107 Midterm Practice Solutions ✔

*Hope that you enjoyed working through any of these problems! I certainly had fun writing them :D — Ben*

### Problem 1: My Zero Academia – Ben’s Life of Funemployment as a Co-Term [Bits and Bytes]

- a. Implement the function **zero\_upper**, which accepts as input a pair of unsigned (32-bit) integers: **value** and **N**. The function should use a bitwise expression to zero out the upper **N** bits of **value**, and return that result. You can assume **N** to be an integer between 1 and 31, inclusive.



For instance, if **value** had binary 11110000111100001111000011110000, and **N** were **8**, then **zero\_upper** should return a value with binary representation

00000000111100001111000011110000

with the upper or most significant 8 bits set to zero.

This is just one possible solution!

```
unsigned int zero_upper(unsigned int value, unsigned int N) {
    /* starts with all 1 bits, shifting right to create a mask with N zero
       bits in front followed by one bits, e.g., for N = 2, 0b00111...111 */

    unsigned int top_zeroes = -1U >> N; // U for unsigned
    return value & top_zeroes;
}
```

- b. What does the bitmask expression you wrote above compute numerically about **value**?

**The remainder when value is divided by  $2^{(32 - N)}$ .**

The reason: polynomial division! Observe that bits are coefficients of a binary polynomial (e.g., 0101 is  $0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ ). Note also that **zero\_upper()** will obliterate the higher bits corresponding to higher powers of 2, e.g.,  $2^{31}$ ,  $2^{30}$ , ...,  $2^{(32 - N)}$ . The remaining lower bits, i.e.  $2^{(32 - N - 1)}$ , ...,  $2^0$ , thus represent a remainder, and specifically the remainder at the cutoff bit with power  $2^{(32 - N)}$ .

As a concrete example, take **N = 31**. Zeroing out the upper 31 bits leaves only the LSB: 0 if value is even, and 1 if value is odd. This is identical to the remainder when value is divided by 2, or  $2^{(32 - 31)}$ .

## Problem 2: Life is Like a Box of Chocolates, You Never Know... [Bits and Bytes]

Consider the function **mysteryBox** and its (mysterious, one could say!) implementation below.

```
int mysteryBox(int x, int y) {
    return (x & y) + ((x ^ y) >> 1);
}
```



a. What does the following code print?

```
printf("%d\n", mysteryBox(2, 10));    // %d is placeholder
printf("%d\n", mysteryBox(10, 20));   // to print out an integer
printf("%d\n", mysteryBox(20, 60));   // value, not its binary!
```

The code will print:

```
6
15
40
```

b. (★ **Optional Fun Challenge**) In general, what mathematical relationship or property between any integers **x** and **y** does **mysteryBox(x, y)** compute? Why does it work?

**mysteryBox** computes the midpoint between the two integers **x** and **y**.

To see how its bitwise expression gathers the needed powers of 2 of the midpoint, first recognize that the bits are coefficients of a binary polynomial (e.g., **0101** is  $0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ ). Now consider that:

- **x & y** picks out the non-zero coefficients (1-bits) which are shared
- **x ^ y** picks out the non-zero coefficients which are not shared between the two operands.

For averaging, the **non-shared bits need to be divided by 2 (i.e., right-shifted)**, whereas **the shared coefficients do not need to be divided by 2 because there are two copies of them in the sum**.

We add the bit groups together to produce the midpoint, allowing powers to carry over if needed (*from the Lab 1 extra problem solutions*).

**Trivia:** This expression is immune to both positive and negative overflow!



### Problem 4: The Power of Friendship / Has Bit Friends [Bits and Bytes]

Implement a function `hasBitFriends`, which accepts a 32-bit integer `num` as input, and returns `true` if the binary representation of `num` has a pair of consecutive 1 bits, and `false` otherwise.

**F.R.I.E.N.D.S**

This was my one-liner approach 😊.

```
bool hasBitFriends(int num) {
    return num & (num << 1) != 0;
}
```

**Explanation:** If `num` has consecutive 1 bits in its binary, when we shift `num`'s bits to the left by one position, there should be overlapping 1 bits with the original. If there were no consecutive 1 bits, no overlap would exist.

The expression `num & (num << 1)` then evaluates to 0 when no 1 bits overlap between `num` and `num << 1`, and non-zero if we have some overlap. Alternatively, this could be written with the (unsigned / logical) right shift as

```
return num & ((unsigned int) num >> 1) != 0;
```

We must cast to **unsigned**, as right-shifting a signed, negative value will fill the MSB with 1 bits / imaginary friends.

Here's a (less elegant, though commendable) approach that loops over each neighboring pair of bits.

```
bool hasBitFriends(int num) {
    const int INT_BITWIDTH = 32; // 32-bit integers in C

    /* checks pairs of neighboring bits going from left to right */
    for (int i = 0; i < INT_BITWIDTH - 1; i++) { // 31 pairs for 32 bits

        /* checks if bits i, i + 1 form a consecutive pair of 1 bits */
        int both_ones = 0b11 < (INT_BITWIDTH - 2 - i);
        if (num & both_ones == both_ones) {
            return true;
        }
    }

    return false; // no pair found
}
```

### Problem 5: Life Comes Full Circle [Strings]

Implement the function `comesFullCircle`, which accepts a `char* str` and an integer `N` as input, and returns `true` if the string `str` has a matching prefix and suffix that is precisely `N` characters long, and `false` otherwise. That is, we return true only if the first `N` characters matches the last `N` characters, given position, and return that result.

Some examples are shown below:

- `comesfullCircle("hunterXhunter", 3)` should return `false`.
- `comesfullCircle("hunterXhunter", 6)` should return `true`.
- `comesfullCircle("lalala", 2)` should return `true`.
- `comesfullCircle("lalala", 4)` should return `true`.

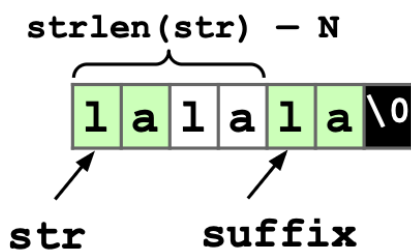


Be sure to take advantage of the `<string.h>` library functions. You can assume `N >= 1`, but in the possible event that `N` exceeds the string's length, you should simply return false.

```
bool comesFullCircle(char* str, int N) {
    if (N > strlen(str)) {
        return false; // can't have prefix or suffix longer than string
    }

    char* suffix = str + (strlen(str) - N); // points toward last N chars

    // compares the first N chars (prefix) with last N chars (suffix)
    return strncmp(str, suffix, N) == 0;
}
```



N 2 full circle? ✔

## Problem 6: String Theory, From All Matter to Strings 🍌 [Strings]

The `atoi` function converts a string of digits to a decimal, but no standard function exists that converts in reverse. Here, we'll implement the function `iota`, which will accept as input an **unsigned long num** and a **char[] buffer**. The function should convert `num` into a string of digits, and store the result in the passed-in buffer. You can assume that `buf` holds enough space for each digit character and null-terminator `\0`, and that `num` is a positive value.

For instance, calling `iota(107, buf)` should populate `buf` as the string of digits `"107"`. Note also that the expression `'0' + digit` will get the character that corresponds to an integer digit.

Here's an approach that first computes how many digits are in the number, to see how far into the buffer that we'll have to write into. Note the units digit is written furthest down, and the leftmost digit at index 0.

```
void iota(unsigned long num, char[] buffer){
    /* get how many digits in num, using a pretty cute for loop */
    int digit_count = 1;
    for (int remainder = num; (remainder /= 10) > 0; digit_count++);

    /* iterates through digits from right to left, adding to buffer */
    buffer[digit_count] = '\0';
    for (int i = digit_count - 1; i >= 0; i--){
        int curr_digit = num % 10; // get digit on right
        buffer[i] = '0' + curr_digit;
        num /= 10;                // shave off current rightmost digit
    }
}
```

Here's an approach that first writes the digits to buffer in reverse order, then swaps them back to normal.

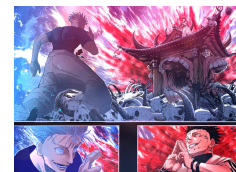
```
void iota(unsigned long num, char[] buffer){
    /* first, add the digits to the buffer in reverse order */
    int len = 0;
    while (num > 0){
        buffer[len] = '0' + (num % 10); // get digit on right
        num /= 10;                     // shave off current rightmost digit
        len++;
    }

    /* then perform swaps to reverse the elements in the buffer */
    for (int i = 0; i < (len - 1) / 2; i++){
        char temp = buffer[i];
        buffer[i] = buffer[len - 1 - i];
        buffer[len - 1 - i] = temp;
    }

    buffer[len] = '\0'; // don't forget the null-terminator!
}
```

## Problem 7: Domain Expansion, Malevolent Heap [Dynamic Memory]

Implement a function **expand**, which receives an array of `char* strings`, and an integer `strings_count` as input. The function should return a string which is the concatenation of all `strings_count` strings in `strings`. This returned string should be heap-allocated, and it's the client's responsibility, not that of **expand**'s implementation, to free this memory when done.



For instance, if `strings` consists of `"cleave"`, `"dismantle"`, `"divine"`, and `"flame"`, with `strings_count` equal to 4, then **expand** should return a heap-allocated string `"cleavedismantledivineflame"`. Your implementation should not free any of the strings in the input array, and don't forget space for the null-terminator!

```
char* expand(char* strings[], int strings_count){
    if (strings_count == 0) return strdup(""); // not required by problem

    /* start with first string, creating a heap-allocated copy */
    char* result = strdup(strings[0]);

    /* loop through other strings, concatenating each with result */
    for (int i = 1; i < strings_count; i++){
        size_t new_len = strlen(result) + strlen(strings[i]) + 1;
        result = realloc(result, new_len); // enlarge heap buffer
        strcat(result, strings[i]);
    }

    return result;
}
```

Here's an alternative that doesn't use `realloc`, but needs to loop through the array of strings twice.

```
char* expand(char* strings[], int strings_count){
    /* figure out length of eventual concatenated result */
    size_t final_len = 0;
    for (int i = 0; i < strings_count; i++){
        final_len += strlen(strings[i]);
    }

    /* create large enough heap buffer, then join together all strings */
    char* result = malloc(final_len + 1); // +1 for the null-terminator
    result[0] = '\0'; // allows initial concatenation with strings[0]
    for (int i = 0; i < strings_count; i++){
        strcat(result, strings[i]);
    }
    return result;
}
```

### Problem 8: Domain Expansion II, Unlimited Void Stars [Dynamic Memory]

Write a (simplified) implementation of `calloc`, which dynamically allocates an array of `nelems` elements each of size `elemsz` bytes, and returns a pointer to the base address of that array. The memory contents should be all initialized to 0, which is the shining feature that distinguishes `calloc` from ordinary `malloc`, though you are permitted (and should) call the C library function `malloc` in your implementation. Otherwise, you'd need to create your very own heap allocator—don't worry if you don't know what that is just yet! :)



For convenience, you don't have to perform error-checking on the inputs, and you may assume that the total size of the memory region, `nelems * elemsz`, is positive and a multiple of `sizeof(long)` or 8.

```
void *calloc(size_t nelems, size_t elemsz){

    /* first, allocate heap memory for the entire array */
    size_t total_bytes = nelems * elemsz;
    void* array_ptr = malloc(total_bytes);

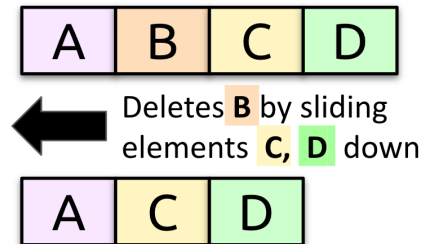
    /* zero out the memory in 8-byte chunks, i.e. size of a long
       could also do smaller byte chunks, but this is more efficient */
    long* chunk_ptr = array_ptr;
    size_t num_chunks = 1 + (total_bytes - 1) / sizeof(long);

    for (int i = 0; i < num_chunks; i++){
        *chunk_ptr = 0;
        chunk_ptr++;
    }

    return array_ptr;
}
```

### Problem 9: Yeet that Element! 🖐️ [Generics]

Implement a generic function **yeet** that, given an array with starting address **base** and a given integer **index**, deletes (or in the colorful language of my generation, “**yeets**”) the element at that **index**. Specifically, we should slide all the elements to the right of the index down, ensuring that the array base should stay in-place. The function is also supplied with the number of array elements passed in as an **int\*** pointer, and the **width** / size in bytes of each element.



We can assume  $0 \leq \text{index} < \text{nelems}$ , and that the array is non-empty. Fill in the blanks below :)

```
void yeet(void *base, int index, int *nelems_ptr, int width){

    void* elemToDelete = (char*) base + index * width;
    void* elemToRight = (char*) elemToDelete + width;

    /* slide all the elements to the right of index down */
    memmove(elemToDelete, elemToRight,
            ((*nelems_ptr) - 1 - index) * width); // in terms of bytes

    /* finally, update the number of elements */
    (*nelem_ptr)--;
}
```

### Problem 10: Comparison Functions are the Thief of Joy [Generics]

- a. Implement a comparison function `cmp_by_last_name`, that would allow the following program to run without crashing, and successfully sort an array of name strings (each comprised of a first and last name, separated by a single space) in **increasing lexicographic order of last name**. People with the same last name can be listed in any order.

```
int main(int argc, char *argv[]){
    char* names[] = {"Chaewon Kim", "Jimin Park", "Tzuyu Chou"};
    size_t num_kpop_stars = 3;
    qsort(names, num_kpop_stars, sizeof(char *), cmp_by_last_name);

    /* names should now be {"Tzuyu Chou", "Chaewon Kim", "Jimin Park"} */
}
```



You can guarantee each string is well-formed, has exactly one space character, and the first and last names are non-empty substrings. You are highly encouraged to use `strchr`, `strcmp`, and/or other string library functions in your implementation.

```
int cmp_by_last_name(void* p_name_one, void* p_name_two){

    char* name_one = *(char **) p_name_one;
    char* name_two = *(char **) p_name_two;

    /* each string is first name, then space character, then last name */
    char* last_name_one = strchr(name_one, ' ') + 1; // + 1 to point
    char* last_name_two = strchr(name_two, ' ') + 1; // after the space

    return strcmp(last_name_one, last_name_two);
}
```

- b. Suppose now that you have a struct defining a K-pop group and the year they were formed, i.e.

```
typedef struct KpopGroup {
    char* name;          /* e.g., Aespa, Le Sserafim, BTS */
    int debut_year;      /* e.g., 2020, 2022, 2013 */
} KpopGroup;
```



Implement a comparison function that could be used (i.e. when passed as a callback function into `qsort`) to sort an array of `KpopGroup` struct elements, from most recent to least recent debut year.

```
int cmp_by_debut_year_reverse(void* p_group_one, void* p_group_two) {

    /* get the structs the input pointers are pointing to */
    KpopGroup group_one = *(KpopGroup *) p_group_one;
    KpopGroup group_two = *(KpopGroup *) p_group_one;

    /* subtract in REVERSE from usual, for descending order of year */
    return group_two.debut_year - group_one.debut_year;

}
```

For instance, the below code should result in moving Le Sserafim (which debuted most recently in 2022, with the song *Fearless*) to the front, Aespa to the middle, and BTS should stay put at the very back.

```
KpopGroup groups[3] = {
    {"Aespa", 2020}, {"Le Sserafim", 2022}, {"BTS", 2013}
};
qsort(groups, 3, sizeof(KpopGroup), cmp_by_debut_year_reverse);
```

**Note:** If we had wanted ascending order of year instead (i.e. {BTS, 2013}, {Aespa, 2020}, {Le Sserafim, 2022}) for the array of groups at the very end), the last line would appear as:

```
return group_one.debut_year - group_two.debut_year;
```

which is the typical **(one) – (two)** pattern for sorting in ascending order, as opposed to the **(two) – (one)** pattern for reverse / descending order.

### Problem 11: Sendai Triple Pointer Domain Clash! [Arrays of Arrays of Strings, ★ Challenge]

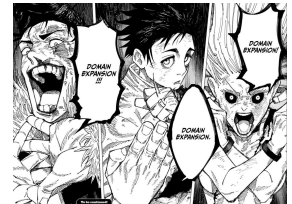
For this fun problem :, you'll draw a memory diagram of the state of memory as it would exist at the end of this code's execution. Make sure to place each item in the correct area (e.g., stack, heap, read-only data).

```
char*** domains = (char ***) malloc(3 * sizeof(char**));
for (int i = 0; i < 3; i++) {
    domains[i] = (char **) malloc((3 - i) * sizeof(char *));
}
char love[9] = "truelove";

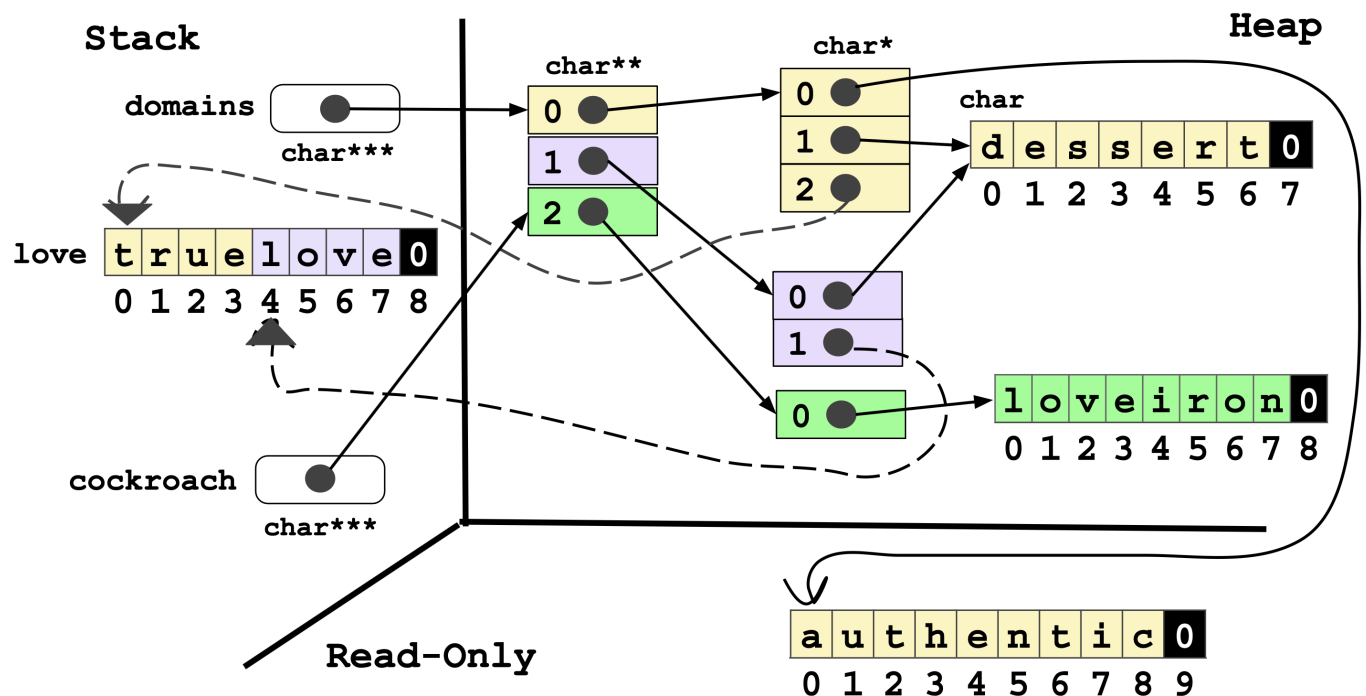
domains[0][0] = "authentic";
domains[0][1] = strdup("mutual!");
domains[0][2] = love;

domains[1][0] = *(domains[0] + 1);
strcpy(*(domains[1]), "dessert");
domains[1][1] = strstr(love, "love");

*(domains[2]) = malloc(sizeof(love));
strcpy(domains[2][0], (char *) love + 4);
char*** cockroach = &(domains[2]);
strcat(**cockroach, "iron"); /* note strcat, not strcpy! */
```



For arrays, make sure to draw boxes and indices for each element (you can skip this for char arrays to save time). Take care to write out any `\0` characters, and check that your pointers are clearly pointing to the correct part of an array. Lastly, if you have any uninitialized memory at the end, you can leave it blank.



**Bonus:** To check your diagram, what strings do `domains[0][1]`, `domains[1][1]`, `domains[2][0]` end up as?

They should be "dessert", "love", and "loveiron", respectively.

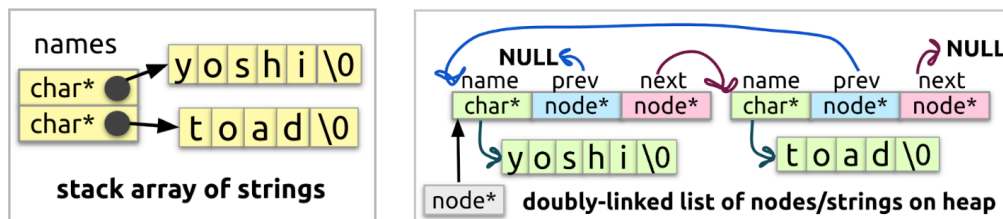
## Problem 12: Switch It Up Like Nintendo 🎮 [Dynamic Memory, Structs, ★ Challenge]

Given a stack-allocated array **names** of valid C strings which represent character names (e.g., Yoshi, Kirby, Zelda, Peach, Pikachu, etc.), implement a function **switch\_it\_up\_like\_nintendo** that converts it into a dynamically-allocated, doubly-linked list of nodes of the following type:

```
typedef struct node {
    char* name;           // pointer for heap-allocated name string
    node* prev;           // pointer to previous node, NULL if none
    node* next;           // pointer to next node, NULL if none
} node;
```

You'll want to do the following in your implementation:

- The linked list should preserve the order of the original array.
- Each node should be heap-allocated, along with each node's name string.
- The **prev** pointer of the starting node, and the **next** pointer of the last node, should be set to **NULL**.
- The function should return a pointer to the starting node.
- If the input array of names is empty (**num\_names = 0**), you should simply return a **NULL** pointer.



**Note:** Arrow (→) is for struct pointers. Use dot (.) if struct type itself, e.g., `(*node_ptr).name`

```
void switch_it_up_like_nintendo(char* names[], int num_names) {

    node* headNode = NULL; /* pointer to starting node */
    node* prevNode = NULL; /* prev of first node is NULL */

    for (int i = 0; i < num_names; i++) {

        /* heap-allocate each node, and its name string */
        node* curNode = malloc(sizeof(node));
        curNode->name = strdup(names[i]);

        /* wire up the current node to linked list */
        if (headNode == NULL) headNode = curNode;
        if (prevNode != NULL) prevNode->next = curNode;
        curNode->prev = prevNode;
        prevNode = curNode;
    }

    prevNode->next = NULL; /* next of last node is NULL */
    return headNode;
}
```

### Problem 13: Beyond the Spider-Verse, and Beyond the Void [Generics, Structs, ★ Challenge]

(Note: Probably clear from its length lol, but this is less of a “short exercise”, more like a fully-fledged exam problem)

Many languages (e.g., Python, JavaScript, Scheme) support heterogeneous arrays, **where elements don’t have to be the same type**—and can in fact be really any type, e.g., integers, strings, nested arrays, all packaged together in one array. By default, C does not support heterogeneous arrays. But with enough effort and patience, we can simulate them with some neat and pointer-based data structures!

As a guiding principle, **each element must be accompanied with information on its type**. For simplicity, we’ll just encode the number of bytes for its data (e.g., 1 byte for a char, 4 bytes for an integer, 8 bytes for a pointer, etc.). Accordingly, the following struct for each element, though lacking flair, suits our needs.

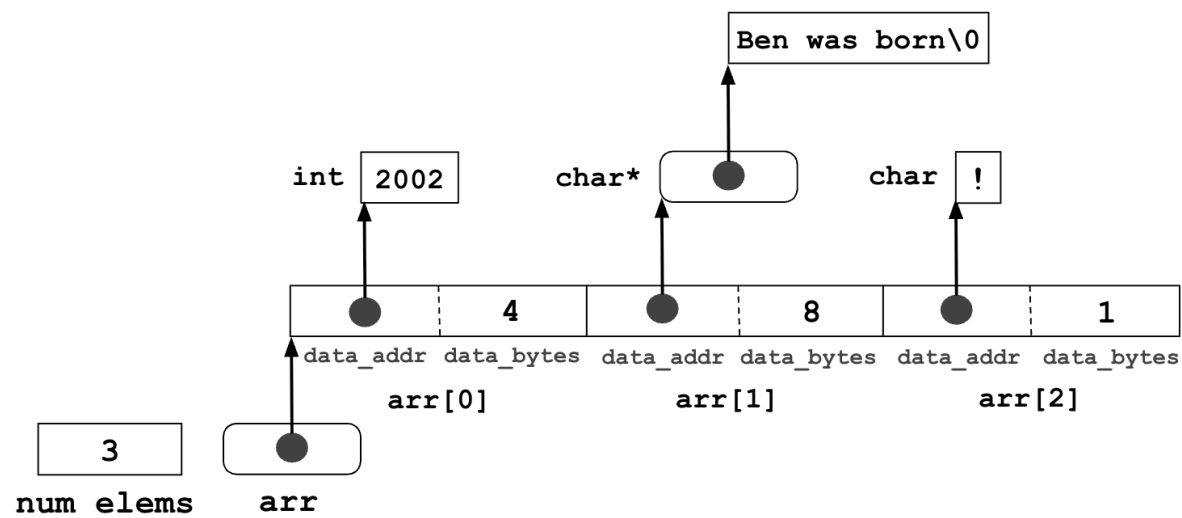
```
typedef struct mixed_elem {
    void* data_addr; /* starting address of where the data is stored */
    size_t data_bytes; /* number of bytes for the element data in memory */
} mixed_elem;
```

And to complete our shining rainbow of a data structure, we use an array of **mixed\_elem** entries. This array should be heap-allocated to allow for resizing, e.g., when needed to accommodate more elements. As an example, this vignette of code below creates a mixed array with pointers to 3 elements—an integer, a string, and a single 1-byte character.

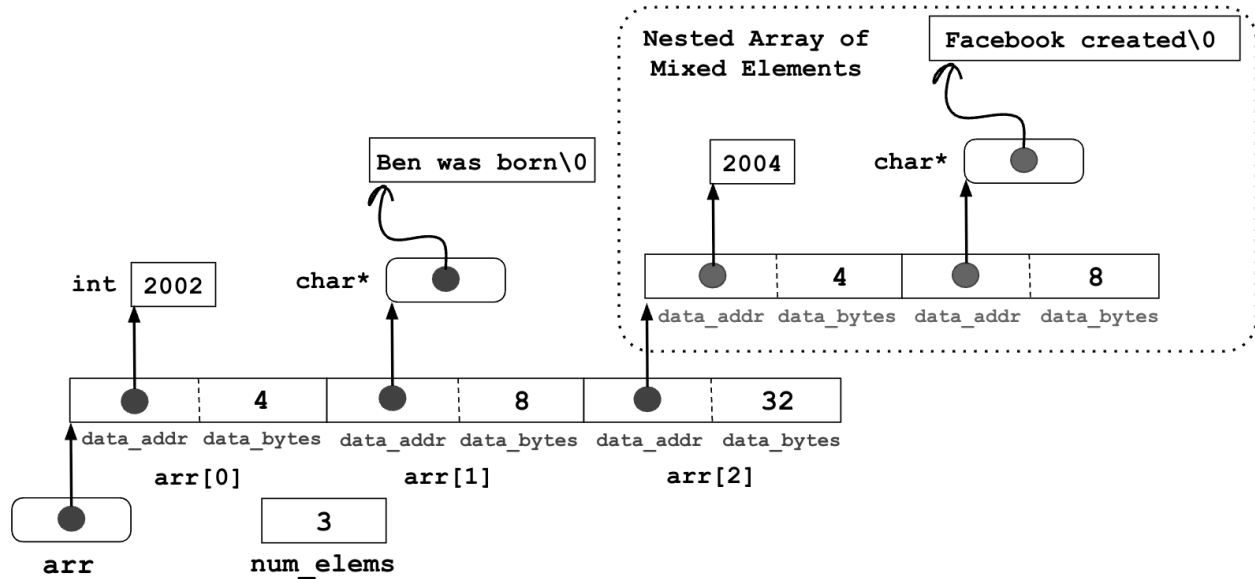
```
int year = 2002;
char* event = "Ben was born"; // wow this guy is unc
char emotion = '!';

size_t num_elems = 3;
mixed_elem* arr = malloc(num_elems * sizeof(mixed_elem));
arr[0] = {&year, sizeof(year)}; /* 1st element is integer */
arr[1] = {&event, sizeof(event)}; /* 2nd element is string */
arr[2] = {&emotion, sizeof(emotion)}; /* 3rd element is character */
```

For reference, a drawing of this heterogeneous array is displayed below.



As promised, we can also support a nested array within this heterogeneous array ☺. For instance, what if instead of a one-character emotion (!) for element `arr[2]`, we had a range of emotions, say another integer and a string, reflecting on what also happened around my birthyear, e.g., the founding of Facebook two years later. The resulting list, **(2002, “Ben was born”, (2004, “Facebook created”))**, with the **(2004, “Facebook created”)** nested, could appear as the following heterogeneous array.



Here, `arr[2].data_addr` points to the first of two elements in the nested array. And as each element is 16 bytes (8-byte pointer and 8-byte `size_t`), then `arr[2].data_bytes = 32` represents that nested array's total width in bytes.

- a. First, you'll implement a function **push\_back** to add an element to the back of an array of **mixed\_elem** structs, enlisting **realloc** to increase the array by exactly one element in size. Since **realloc** may relocate the elements in memory, we pass in the address of the array pointer, hence a double pointer. The array's size is also passed in as a pointer; we should dereference it to update the value accordingly. Lastly, you should also heap-allocate the new element's data, and assign the element's **data\_addr** field to point to that heap copy.

```
void push_back(mixed_elem** arr_ptr, size_t* num_elems_ptr,
               void* elem_data_addr, size_t elem_data_bytes) {

    /* enlarge the array by one struct element (void* pointer + size_t) */
    size_t new_index = *num_elems_ptr;
    *arr_ptr = realloc(*arr_ptr, (new_index + 1) * sizeof(mixed_elem));

    /* heap-allocate a copy of the new element's data (#twinning) */
    void* doppelganger_data = malloc(elem_data_bytes);
    memcpy(doppelganger_data, elem_data_addr, elem_data_bytes);

    /* add the element to the array, referring to the heap copy's data */
    mixed_elem* arr = *arr_ptr;
    arr[new_index].data_addr = doppelganger_data;
    arr[new_index].num_bytes = elem_data_bytes;

    /* lastly, increment the counter for the array's size */
    (*num_elems_ptr)++;

}
```

- b. Next, you'll implement a function to remove the element at a specific index of an array of **mixed\_elem** structs, by sliding all the elements to the right down. You are more than welcome to call the **yeet** function from Problem 9 ☺, and should free the element's heap data pointed to by **data\_addr** first. And if the supplied **index** exceeds the array bounds, you should simply return without doing anything. Lastly, note that we also pass in the address of the array pointer (like the above function), and that you should downsize the array appropriately with **realloc**.

```
void remove(mixed_elem** arr_ptr, size_t* num_elems_ptr, size_t index) {
    if (index >= *num_elems_ptr) return; // when index exceeds bounds

    /* first, free the element data pointed to by data_addr */
    mixed_elem* arr = *arr_ptr;
    free(arr[index].data_addr);

    /* slide the array elements to the right down to remove element */
    yeet(arr, index, num_elems_ptr, sizeof(mixed_elem));

    /* downsize the array, noting that yeet decrements *num_elems_ptr */
    *arr_ptr = realloc(*arr_ptr, (*num_elems_ptr) * sizeof(mixed_elem));
}
```

Alternatively, instead of borrowing the **yeet** function from Problem 9, we could rewrite the element removal logic as follows:

```
size_t num_elems_to_right = *num_elems_ptr - 1 - index;
memmove(&arr[index], &arr[index + 1],
        num_elems_to_right * sizeof(mixed_elem));
(*num_elems_ptr)--; // decrement the counter for the array's size
```

- c. For this part, you will, step by step, implement a **comparison function** **compare\_data\_addr**, which could be used to sort the entries of a heterogeneous array based on **each element's data address, in increasing order**. Note that the two input parameters, **p\_element\_one** and **p\_element\_two**, are **pointers to mixed\_elem** structs.

```
int compare_data_addr(void* p_element_one, void* p_element_two) {
    void* one_data_addr = _____ Blank A _____;
    void* two_data_addr = _____;
    return _____ Blank B _____;
}
```

That is, a line such as the below, using the variables defined and manicured in the code snippets above,

```
qsort(arr, num_elems, sizeof(mixed_elem), compare_data_addr);
```

should run without crashing, and should re-arrange the (non-nested) structs in-place to ensure the array is in sorted order.

**For blank A, circle all expressions** (select all, sorry ☺) that would **correctly fetch the address of the 1st element's data**. You can trust the code for getting the 2nd element's address is similar. As a tip, draw things out! And note that in the `mixed_elem` struct, the first 8 bytes store the `void*` address of where that element's data begins in memory.

|                                        |                                                         |
|----------------------------------------|---------------------------------------------------------|
| <code>(p_element_one).data_addr</code> | <code>(*(mixed_elem *) p_element_one).data_addr;</code> |
| <code>(void *) p_element_one</code>    | <code>*(void *) p_element_one;</code>                   |
| <code>*(void **) p_element_one</code>  | <code>** (void ***) &amp; p_element_one;</code>         |

Though disguised as a `void*` input parameter, `p_element_one` points to a mixed struct element, so it's a `mixed_elem*` pointer type. However, it moonlights also as a `void**` pointer, as the first 8 bytes of the struct (which `p_element_one` points to) stores the `void*` pointer we eventually want to access. Broadly, a `void**` is a pointer that points to a `void*`, i.e. a pointer to a pointer of unspecified or any type.

- `*(mixed_elem *) p_element_one).data_addr` works as we cast to a correct type, then dereference to get the struct itself, which has a `data_addr` field to fetch the element data address.
- `*(void**) p_element_one` works as this fetches the first 8 bytes of the memory address that `p_element_one` points to, which is the `void*` data address inside the struct that we want!
- `** (void ***) & p_element_one` is mischievous and very cursed, though it still works! Since `p_element_one` is a `void**` pointer, taking its address leads to a `void***` (triple star). We can dereference it twice to get the `void*` that `p_element_one` was pointing to, or the data address ☺

**For blank B, circle the one expression** that would correctly obtain the **difference in bytes between the addresses of element data**.

|                                                                |
|----------------------------------------------------------------|
| <code>(char *) one_data_addr - (char *) two_data_addr</code>   |
| <code>*(char *) one_data_addr - *(char *) two_data_addr</code> |
| <code>one_data_addr - two_data_addr</code>                     |
| <code>*one_data_addr - *two_data_addr</code>                   |

For `void*` pointers, we cast to a `char*` to perform pointer arithmetic in terms of bytes.

- d. Next, implement a **comparison function** `compare_data_size`, which could be used to sort, in increasing order, the struct entries of a heterogeneous array based on **each element's data size (in bytes) instead**. Note that again, the input arguments are pointers to `mixed_elem` struct elements.

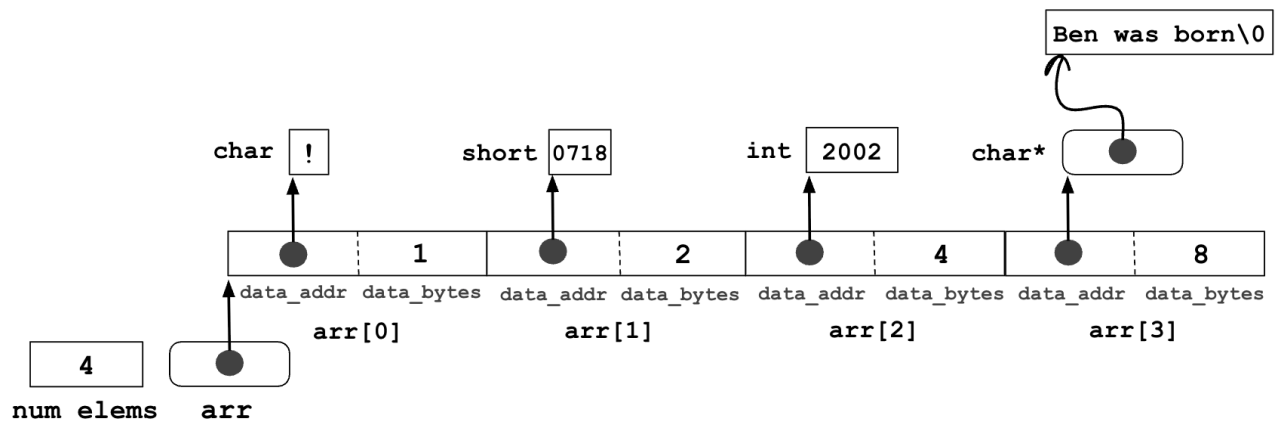
```
int compare_data_size (void* p_element_one, void* p_element_two) {

    /* note that p_element_one, p_element_two are POINTERS
       to struct elements, not the elements themselves */

    mixed_elem one = *(mixed_elem *) p_element_one;
    mixed_elem two = *(mixed_elem *) p_element_two;

    return one.data_bytes - two.data_bytes;
}
```

The following array would be considered sorted by data size, which can be convenient for grouping elements together.

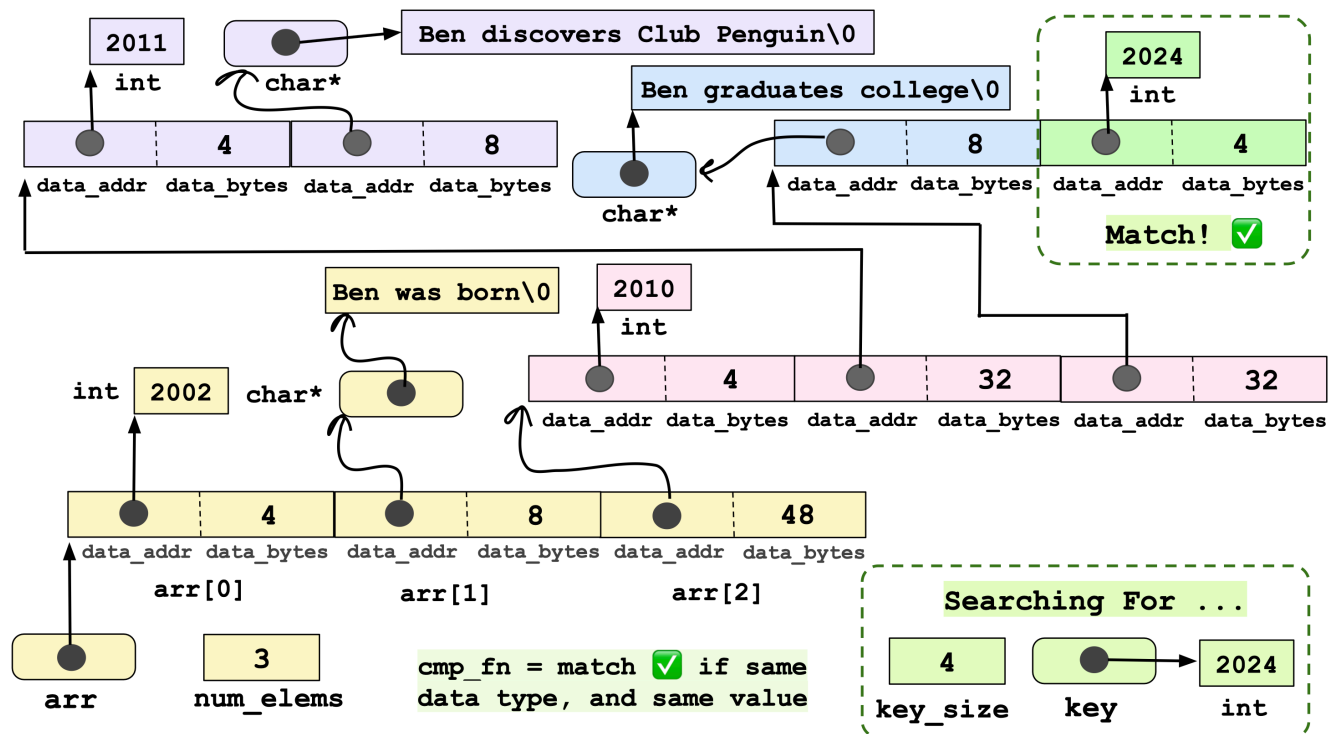


e. ★ Ben's Bonus Wicked Final Problem (And Because Why Not, Recursive Backtracking is Back! ☺)

For this part, assume that elements in the heterogeneous array are restricted to chars (1 byte), shorts (2 bytes), integers (4 bytes), strings (8 bytes, as the size of the char\* pointer), and nested **mixedElem** arrays ( $\geq 16$  bytes, and a multiple of 16 bytes in size, as each **mixedElem** is 16 bytes). Note that size fully determines type.

Write a function **findElement** that accepts as input, a heterogeneous array **arr** (with **num\_elems** elements at the current layer), and a **key** (pointer to data of **key\_size** bytes) to search for. The function should **return a pointer to an array element (at any arbitrarily nested depth) that matches the key**—meaning that they are considered equal under the supplied comparison function **cmp\_fn**, which accepts as input a pair of pointers to **mixedElem** structs. If no such matching element is found, or if **arr == NULL**, simply return a **NULL** pointer. If there are multiple possible matches, returning a pointer to any one of them is perfectly acceptable.

In the example illustrated below, we have the heterogeneous array (2002, "Ben was born", (2010, (2011, "Ben discovers Club Penguin"), ("Ben graduates college", 2024) )), with two layers of nesting! With **cmp\_fn** checking for same type and same value, a matching element was found on the final layer, as highlighted.



You can guarantee the input key points to either a char, short, integer, or a string. Hence, when iterating over the elements of any heterogeneous array, if an element's size is  $\geq 16$ , you can guarantee it's a nested array, and should also search it recursively for the key. Otherwise, use **cmp\_fn** directly to check for a match.

You'll present your implementation on the next page—best of luck, and I believe in you!!

```

mixed_elem* findElement(mixed_elem* arr, size_t num_elems, void* key,
                        size_t key_size, int (*cmp_fn) (void *, void *)) {

    /* return a pointer to a matching element, or NULL if none exists */

    if (arr == NULL) return NULL;

    mixed_elem key_to_find = {key, key_size};
    const size_t MIXED_ELEM_SIZE = sizeof(mixed_elem); // 16 bytes

    /* scan through all elements in array (current layer) for a match */
    for (size_t i = 0; i < num_elems; i++) {
        size_t elem_size = arr[i].data_bytes;

        if (elem_size < MIXED_ELEM_SIZE) {
            /* either a char, short, int, string, so check directly */
            if (cmp_fn(&key_to_find, &arr[i]) == 0) {
                return &arr[i];
            }
        } else {
            /* instead points to a nested array of mixed elements */
            mixed_elem* nested_arr = (mixed_elem *) arr[i].data_addr;
            size_t num_nested_elems = elem_size / MIXED_ELEM_SIZE;

            /* recursively search the nested array for the key */
            mixed_elem* match = findElement(
                nested_arr, num_nested_elems, key, key_size, cmp_fn);

            if (match) return match; // end the search if match found!
        }
    }

    return NULL; // no matching element found!
}

```