

CS107 Midterm Practice Problems

This basket of problems is intended to be less elaborate and poetic than problems you can expect on the midterm, or the practice midterms crafted by Nick, Jerry, and others. But it's a neat way, I think, to **first practice with shorter, focused examples**, testing key concepts (e.g., binary, pointers, strings, arrays, stack & heap memory, generics) and functions **before jumping into the full-blown practice exams!**

These are curated from extra problems on our lab handouts, past review sessions, my tormented imagination, etc., and you can find a table of contents supplied below :)

Problems	Topics
1. My Zero Academia	Bits and Bytes
2. Life is Like a Box of Chocolates	Bits and Bytes
3. Achilles Heel	Bits and Bytes
4. The Power of Friendship / Has Bit Friends	Bits and Bytes
5. Life Comes Full Circle	Strings
6. String Theory, From All Matter to Strings	Strings
7. Domain Expansion, Malevolent Heap	Dynamic Memory
8. Domain Expansion II, Unlimited Void Stars	Dynamic Memory
9. Yeet that Element!	Generics
10. Comparison Functions are the Thief of Joy	Generics
11. Sendai Triple Pointer Domain Clash! ★	Arrays of Arrays of Strings (Char***)
12. Switch It Up Like Nintendo ★	Dynamic Memory, Structs
13. Beyond the Spider-Verse, Beyond the Void ★	Generics, Structs, Everything Really

★ **Note:** Problems 11, 12, 13 are (fun) challenge problems. Problems 4 and 9 are my personal favorites! If you're understandably busy, or looking to prioritize, **I would recommend Problems 1, 4, 5, 7, 9, 10.**

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) {

}
}
```

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.

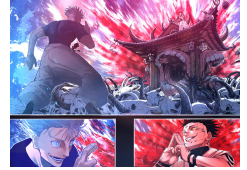
For instance, calling `iota(107, buf)` should populate `buf` as the string of digits `"107"`. Note also that the expression `'0' + digit` will get the character that corresponds to an integer digit.

```
void iota(unsigned long num, char[] buffer) {

}
}
```

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) {
```

```
}
```

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! :)



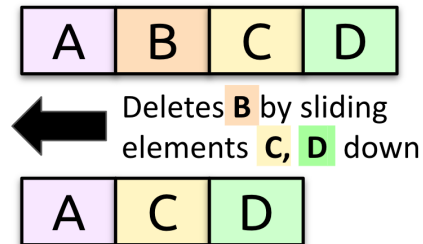
For convenience, you don't have to perform error-checking on the inputs, and you may assume that the total size of the memory region, **nelems * elemsz**, is positive and a multiple of **sizeof(long)** or **8**.

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

```
}
```

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 = _____;
    void* elemToRight = _____;
    /* slide all the elements to the right of index down */
    _____;
    /* finally, update the number of elements */
    _____;
}
```

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) {

}
```

- 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) {

}
```

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);
```

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

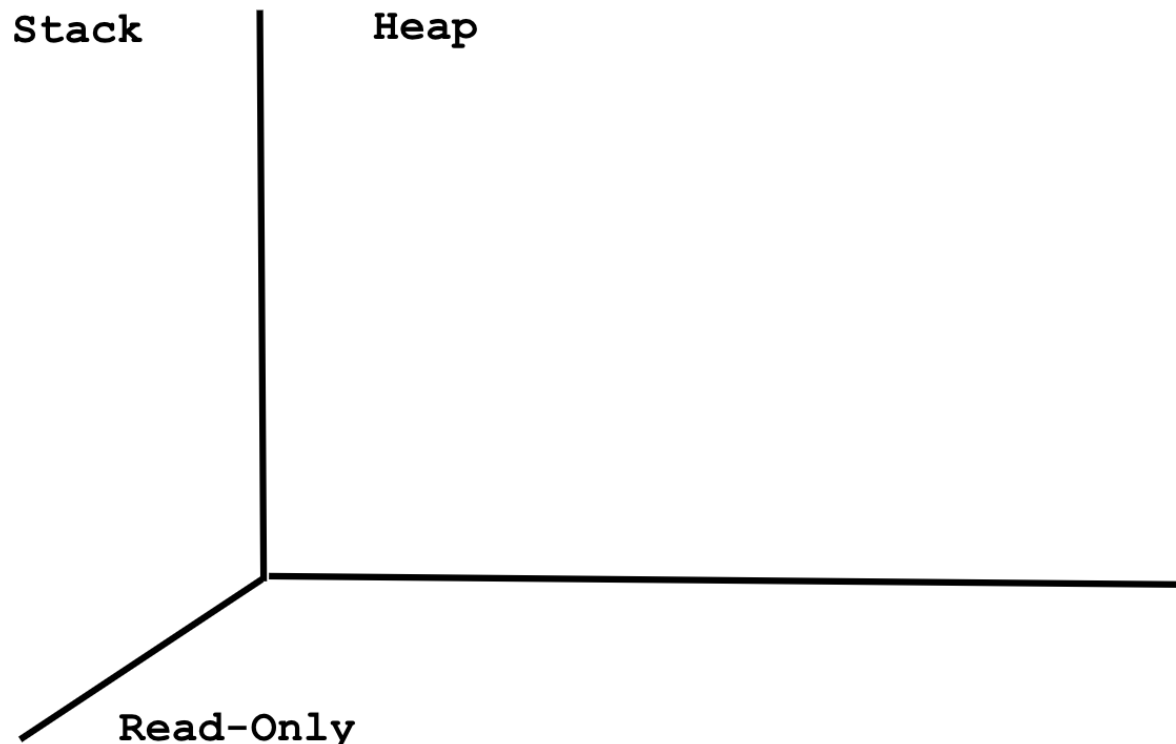
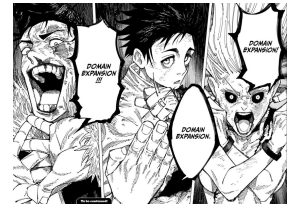
For this fun problem :, you'll draw a memory diagram of the state of memory as it would exist at the end of this code's execution. Make sure to place each item in the correct area (e.g., stack, heap, read-only data).

```
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] = "tru love";

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

domains[1][0] = *(domains[0] + 1);
strcpy(domains[1][1], "dessert");
domains[1][2] = 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! */
```



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

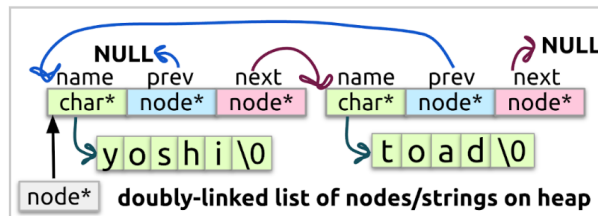
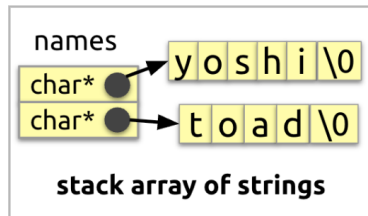
Problem 12: Switch It Up Like Nintendo 🎮 [Dynamic Memory, Structs, ★ Challenge]

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.



```
node *switch_it_up_like_nintendo(char* names[], int num_names) {
    // present your implementation below!
```

```
}
```

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.

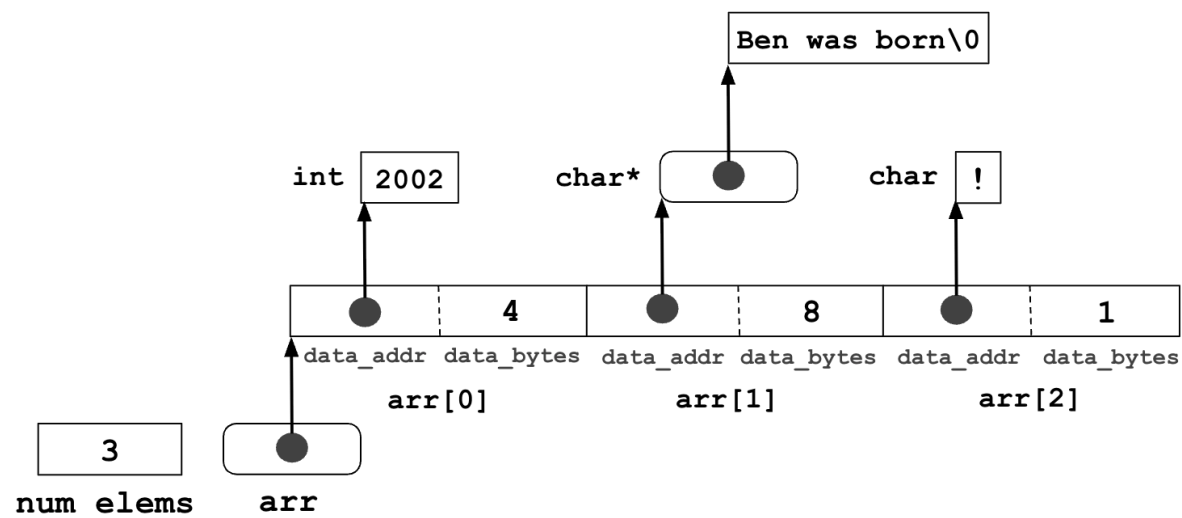
```
typedef struct mixed_elem {
    void* data_addr; /* starting address of where the data is stored */
    size_t data_bytes; /* number of bytes for the element data in memory */
} mixed_elem;
```

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.

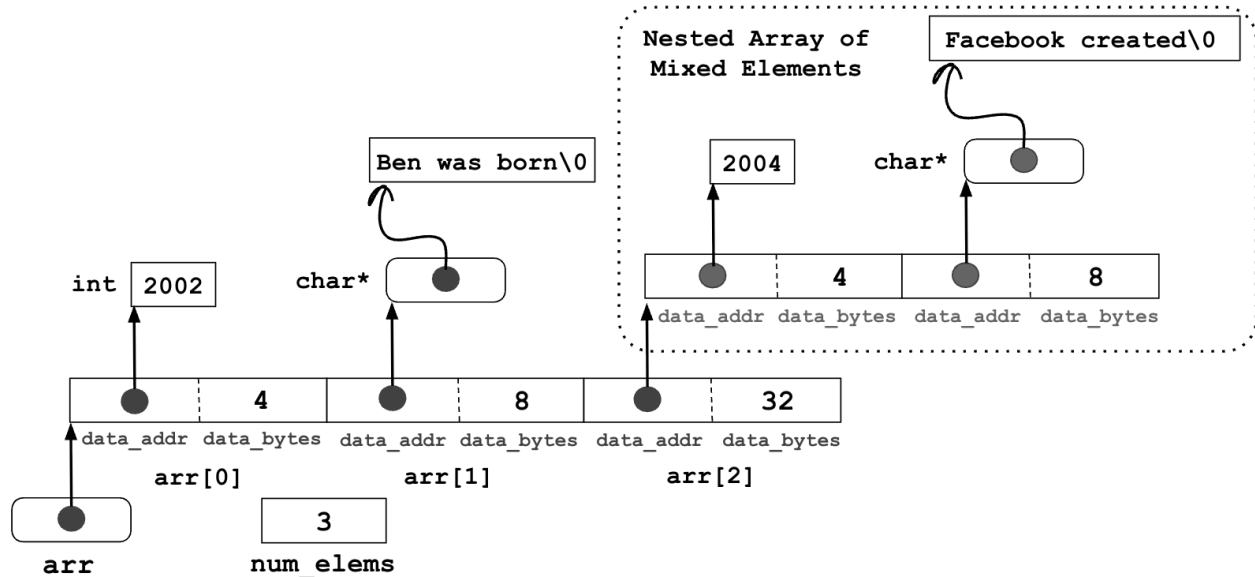
```
int year = 2002;
char* event = "Ben was born"; // wow this guy is unc
char emotion = '!';

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.

- 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) {
```

```
}
```

- 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) {

}

```

- 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.

```
(p_element_one).data_addr      ( *(mixed_elem *) p_element_one).data_addr;
(void *) p_element_one         *(void *) p_element_one;
*(void **) p_element_one       **(void ***) & p_element_one;
```

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

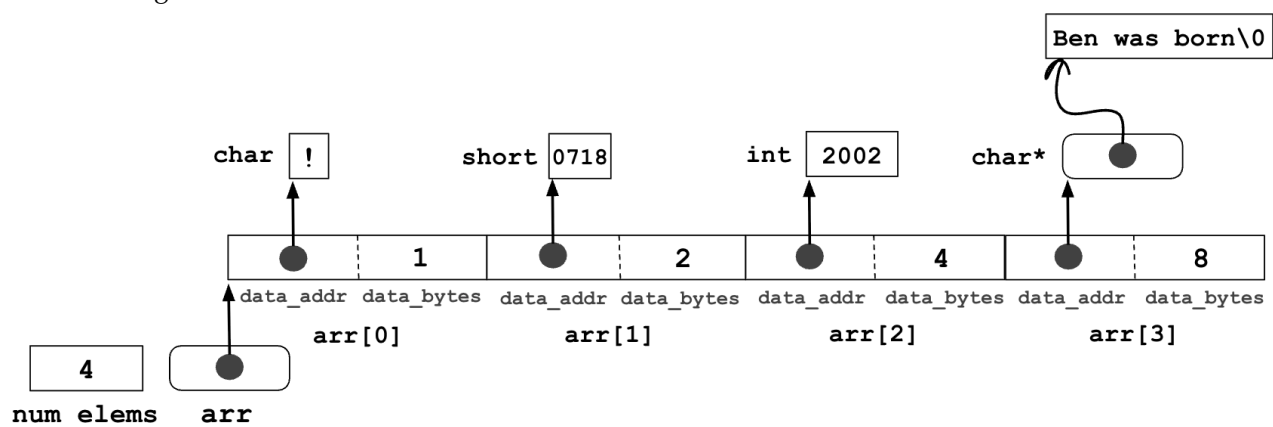
```
(char *) one_data_addr - (char *) two_data_addr
*(char *) one_data_addr - *(char *) two_data_addr
one_data_addr - two_data_addr
*one_data_addr - *two_data_addr
```

- 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) {

}
}
```

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

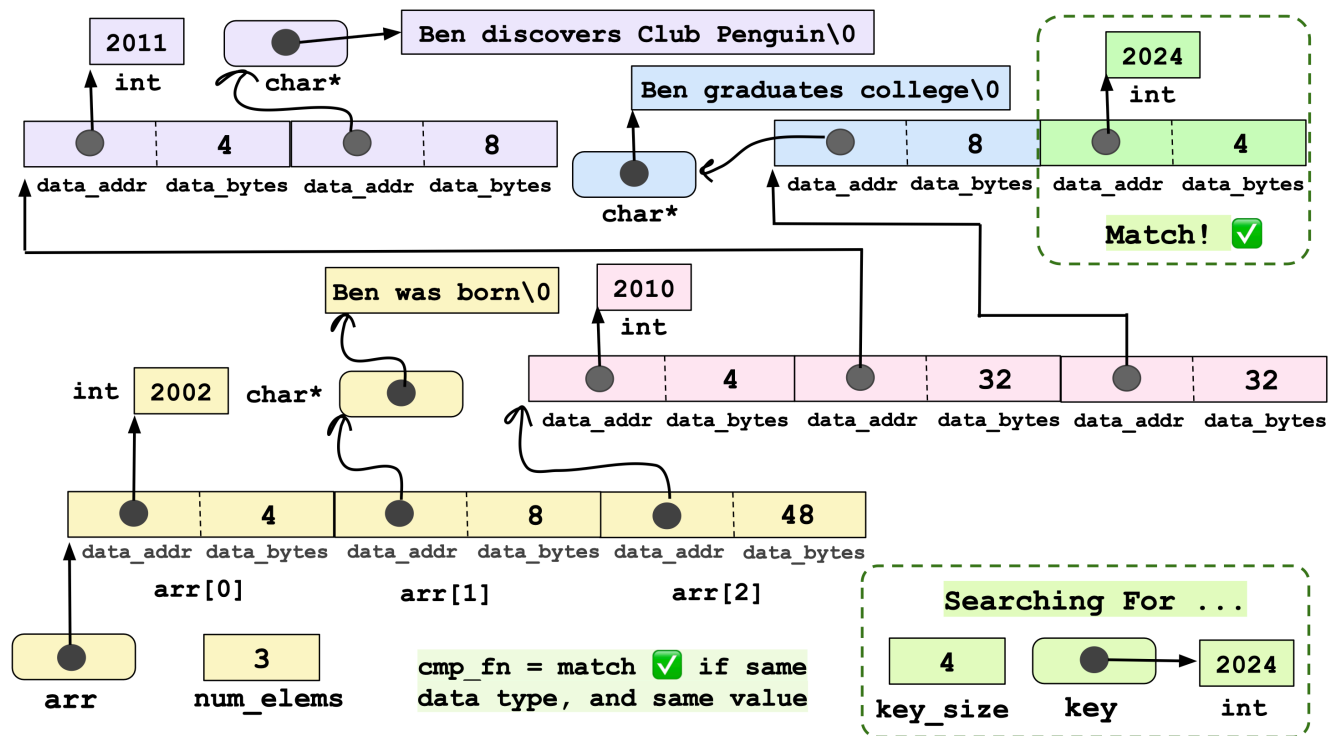


e. ★ Ben's Bonus Wicked Final Problem (And Because Why Not, Recursive Backtracking is Back! ☺)

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 (≥ 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 ≥ 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 */
```

```
}
```