

✅ CS107 Final Practice Solutions ✅

Here's a table of contents for my latest, short and hopefully sweet album of CS107 problems! The theme is exploration of identity and self as my (admittedly long-delayed) graduation approaches this autumn :)

Problems

1. Map of the Soul
2. A Game of Assassins
3. The Disaster Bi
4. Blood, Sweat, and Tears: **realgrind**
5. Gratitude at the End of Ben's Teaching Journey

Topics

Generics
Dynamic Memory
Heap Allocator
Assembly, Reverse-Engineering
Runtime Stack, Hacking the Stack

Thanks everyone – you're going to do wonderful on the final! ❤️



Problem 1: Map of the Soul 💖 [Generics]

[a] Implement a generic function **map**, which applies an input function to each element in an array. For instance, we could convert an array of integer temperatures in Celsius to Fahrenheit (by multiplying each integer by 9/5 and adding 32); or translate each string in an array of *Bangtan Boys* album names from Korean to English (assuming enough space for each potentially longer string / array of English characters).

To be truly generic, the map function accepts four arguments as input, namely:

- **void* arr**, the base address of the array, or equivalently, the address of the starting element.
- **int num_elems**, the number of elements in the array. You may assume **num_elems > 0**.
- **size_t width**, the number of bytes that each element occupies in memory.
- **void (*fn) (void *)**, a function pointer that, given a pointer to an array element, can inflict some change on that element. There is no return value, as evinced by the **void** return signature.

As an example, consider the vignette of code below.

```
int[] arr = {4,5,6};
int num_elems = sizeof(arr) / sizeof(arr[0]);
map(arr, num_elems, sizeof(int), plus_one);
```

The call to **map** should result in the integer array **{4,5,6}** becoming **{5,6,7}**. This of course assumes a fully-functional **plus_one** callback which, given a pointer to an integer, increments its value by 1.

Present your implementation of generic **map** here.

```
void map(void* arr, int num_elems, size_t width, void (*fn)(void *)) {
    /* loop through all n elements, to apply the function fn to each */
    for (int i = 0; i < num_elems; i++) {
        /* get a pointer to the ith element */
        void* elem_ptr = (char *) arr + i * width;

        /* call the supplied function to affect each element */
        fn(elem_ptr);
    }
}
```

[b] Suppose we indeed have an array of integers. Write a function **plus_one** that can be passed into your generic **map** function above (along with the array), that would result in incrementing each element's value by 1. Your function should not allocate any new memory, but instead modify the existing integer in-place.

```
void plus_one(void* ptr) {
    (*(int *)ptr)++;    // *(int *) ptr = *(int *) ptr + 1; also valid
}
```

[c] Consider the following struct definition, containing a pointer to a null-terminated buffer of characters storing the song's name, next to an integer count for how many times that song has been streamed.

```
typedef struct song {
    char* name;
    int num_streams;
} song;
```

Now, suppose that you have an array of fully initialized song structs, for instance

```
song new_album[14] = {"Body to Body", 20}, {"Hooligan", 5}, {"Aliens", 3},
                     {"Fy", 5}, {"Swim", 15}, ...}; // assume more elements!
```

Say that you're listening to the full album (*Arirang* by BTS) again, track by track, and thus listening to each song exactly one more time. Write a callback function **play_song** that can be passed into the generic **map** function alongside the array, that would result in incrementing each struct's **num_streams** count by 1.

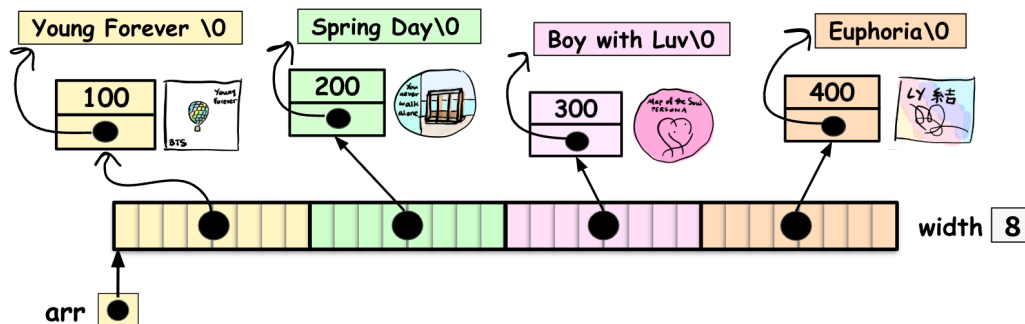
Note that we need to cast the **void*** pointer to a **song***, as **ptr** is pointing to a **song** struct.

```
void play_song(void* ptr) {
    (*(song *) ptr).num_streams++;
}
```

/* equally valid approach if you'd prefer to use the arrow -> notation, i.e.
 * if we have a pointer to a struct, (*ptr).field is the same as ptr->field
 */

```
void play_song(void* ptr) {
    ((song *) ptr)->num_streams++;
}
```

[d] Using the same **song** struct above, suppose now that you have an array of **pointers to song structs**. That is, each element in the array is a pointer, holding the address of some song in memory. Your task? Write a new implementation of **play_song**, aptly named **play_song_again**, that can be passed into the generic **map** alongside the array of 8-byte pointers, to ultimately increment each struct's **num_streams** count by 1.



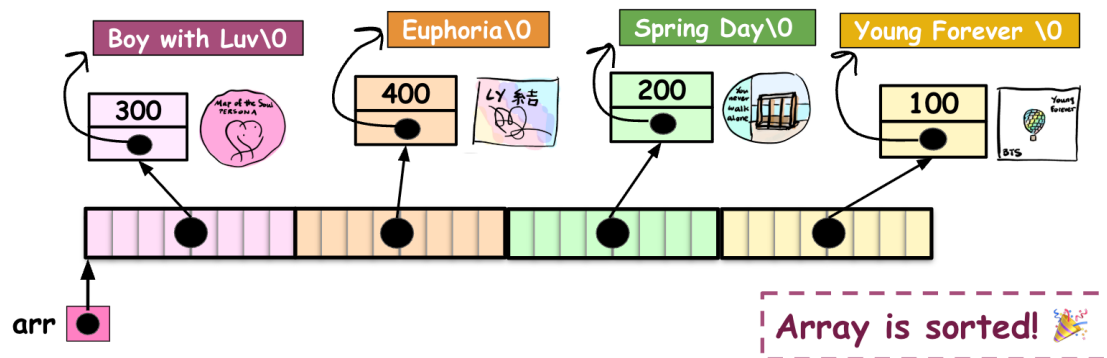
A diagram is provided above for reference, illustrating an array of 4 pointers, each to a unique **song** struct.

Note: The hand drawn album sketches are purely for aesthetic purposes, and are not actually in memory 🙄.

Note that we need to cast the `void*` to a `song**` pointer. Each array element is a `song*` pointer, and since we pass in pointers to the element, then we get a double pointer passed into `play_song`.

```
void play_song(void* ptr) {
    (**(song **) ptr).num_streams++;
}
/* alternatively, we could do (*(song **) ptr)->num_streams++; */
```

[e] Similar to the previous subproblem, assume that you have an array of pointers to song structs. You're interested in sorting the pointer elements so that the structs they point to are organized alphabetically by the name of each song. For the illustrated example in part [d], a sorted version of the array would appear as:



Here, the pointer element corresponding to the “**Boy with Luv**” song is first, followed by the pointer elements corresponding to “**Euphoria**”, “**Spring Day**”, and “**Young Forever**” songs in that order.

You wish to enlist `qsort` to accomplish this, and need to pass in a comparison function that, given pointers to array elements, can ultimately compare the song names by increasing lexicographic order. Present your implementation of `cmp_by_song_name` below. You are highly encouraged to use library function `strcmp`.

```
int cmp_by_song_name(void* one, void* two){
    song song_one = **(song **) one;
    song song_two = **(song **) two;
    return strcmp(song_one.name, song_two.name); /* name is char* field */
}
```

Alternatively, we can use the (remarkable) fact that `one` and `two` both moonlight as triple pointers :D to characters (see the diagram above, note it takes 3 pointer hops from `arr` to ``B``), or `char***` pointers.

```
int cmp_by_song_name(void* one, void* two){
    char* name_one = **(char ***) one; // dereference twice to get char*
    char* name_two = **(char ***) two;
    return strcmp(name_one, name_two);
}
```

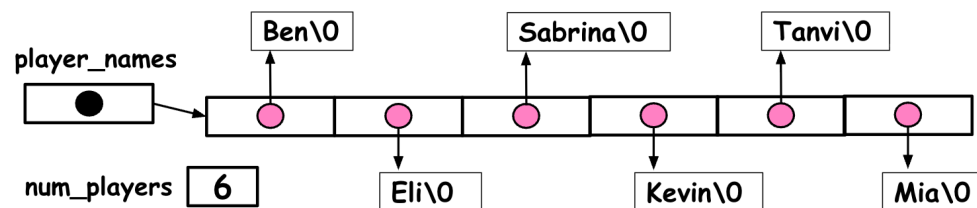
Problem 2: A Game of Assassins [Dynamic Memory]

Assassins is a popular game on college campuses and dorms, where each student in the game is assigned another student (sometimes even a close friend ❤️) as their “target”, meaning they should eliminate them with a toy weapon. A water pistol, or the metaphorical stab in the heart that is a betrayal tends to suffice.

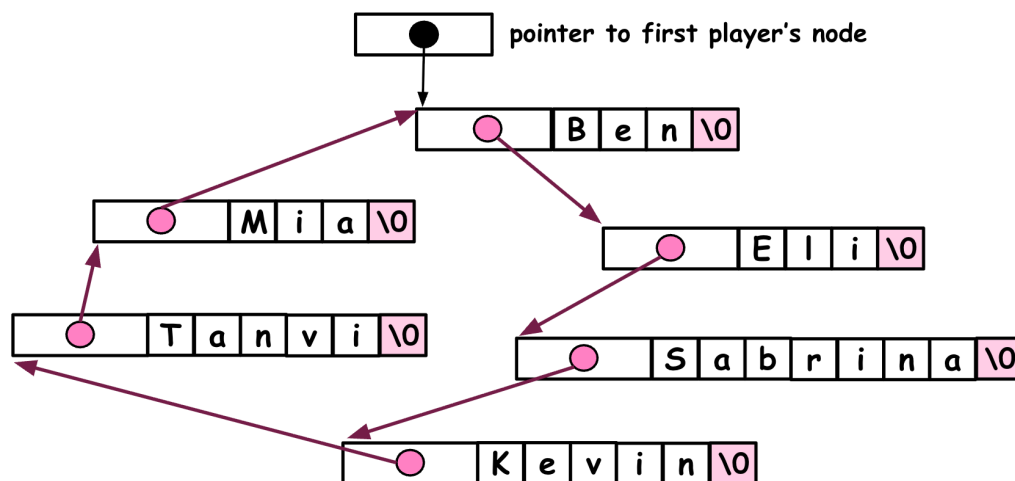
The sequence of target assignments ultimately forms a closed loop or chain, where player A targets player B, player B targets player C, and so forth, with the last player targeting player A. When a player eliminates their target, they adopt the eliminated player’s target as their own going forward. Eventually, the game contracts down to two players who have each other as targets—and whoever eliminates the other first is the winner!

In fact, the flock of CS section leaders recently held a game of assassins on campus, which I partook in. The result? Being ignominiously slimed out on Day 2, not one minute after my CS107 office hours ended. Pooh.

For this problem, say that you’re the game master. To prepare for a game of assassins, you are provided with a list of unique player names, represented as an **array of (at least two) null-terminated strings**, for instance:



You want to write a function `create_assassins_game`, which takes the array of names above, and constructs a **circular, singly-linked list of perfectly-sized, heap-allocated name nodes**, as illustrated below. The first player (Ben) should have the second player (Eli) as their target, and point to their node; the second player (Eli) should have the third player (Sabrina) as their target, pointing to their node, and so forth. The last player (Mia) should then point to the first player (Ben), completing the circular chain of vibrant destruction.



Each variably-sized node is composed exclusively of an **8-byte `void*` pointer to the next node, wedged next to the player’s name represented not as an additional pointer, but as an array of characters ending in a null-terminator**, with no padding between the pointer and starting character of the array.

The function should also return a pointer to the node corresponding to the first player in the input array of names ("**Ben**" in the example above). Use this page to provide your implementation.

```
void* create_assassins_game(char*[] player_names, size_t num_players) {

    void* front_node = NULL; // pointer to first player's node
    void* prev_node = NULL;  // lagging pointer to previous player's node

    /* loop through all players in the game, creating nodes for them */
    for (size_t i = 0; i < num_players; i++) {

        /* creates a perfectly sized node, and copies the player's name
           into the memory region past the first 8 bytes */
        void* curr_node = malloc(sizeof(void*)
                                + strlen(player_names[i]) + 1);
        strcpy((char *) curr_node + sizeof(void*), player_names[i]);

        if (front_node == NULL) front_node = curr_node;

        /* sets first 8 bytes of previous node to point to current one */
        if (prev_node != NULL) *(void **) prev_node = curr_node;

        prev_node = curr_node;
    }

    *(void **) prev_node = front_node; // last player points back to front
    return front_node;
}
```

P.S. Apologies to the amazing Eli Browne! Was nothing personal, but you were my first target in game 😓.

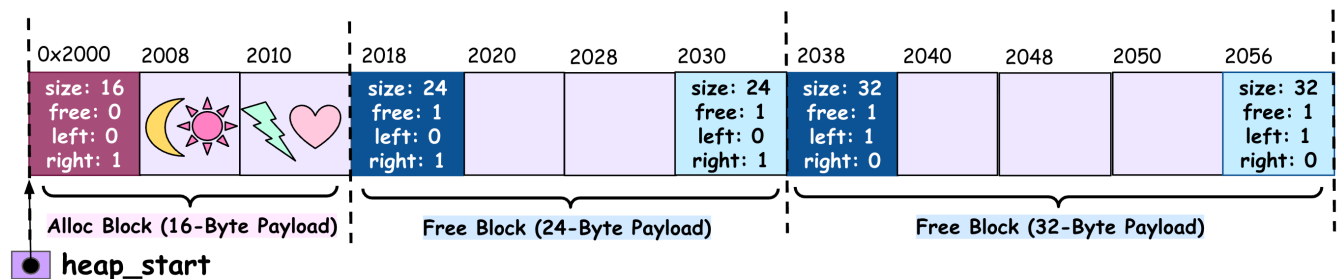
Problem 3: The Disaster Bi 🌈 [Heap Allocator]

Every winter and spring quarter that Ben works as a TA for CS107, he builds a new heap allocator. This says a lot about the state of his life, but I digress. Last quarter, it was a binary search tree allocator with three pointers per free block (left child, right child, parent node) that was designed to support malloc with best-fit search in logarithmic time. This quarter, inspired by the celebration of one's messy self, and my favorite works of young adult fiction, I'll (informally) call my newest, most chaotic allocator **the disaster bi**.



Consider an implicit (though disastrously indecisive, as you'll find out in later part ☺) heap allocator, with the following lovely properties—intended to support coalescing in both directions with an added footer.

- Each block has an 8-byte header. The MSB records if the block is free (1 free, 0 in-use).
- Additionally, the 2nd MSB of the header records if the block to the left is free (1 free, 0 otherwise), and the 3rd MSB records if the block to the right is free (1 free, 0 otherwise). The lower 61 bits encode the size of the payload in bytes. For instance, a 24-byte-payload block that is free, with left and right neighbors allocated, would have a header whose bit representation is 0b10000...0011000.
- Free blocks have an 8-byte footer that overlays the last 8 bytes of its payload. The encoding scheme for its bits mirrors that of its corresponding header: 3 bits for allocation status, the rest for payload.
- All blocks are aligned on 8-byte boundaries. All request sizes are rounded up to the nearest multiple of eight bytes, and each address returned by or gracefully handled by **malloc/free/realloc** is aligned accordingly. The minimum payload size in this allocator is 16 bytes.



In the above miniature example, the 16-byte payload with base address 0x2000 is allocated. It has no left neighbor and a free right neighbor; and speaking of which, the 24-byte payload with base address 0x2018 is indeed free. To its right, the 32-byte payload with base address 0x2038 is free, with no right neighbor.

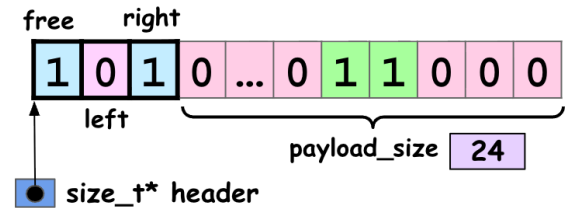
Assume the following **#define** constants and global variables have already been setup and initialized.

```
#define HEADER_SIZE 8
#define FOOTER_SIZE HEADER_SIZE
#define ALIGNMENT 8
#define MIN_PAYLOAD_SIZE 16
```

```
static size_t* heap_start; /* base address of the entire heap segment */
static size_t heap_size; /* number of bytes in the entire heap segment */
```

Part A: A Bit of Help Please! 🙏 (Header/Footer Helper Functions)

First, you'll write an ensemble of nifty helper functions for accessing the dense information stored in the headers (and footers!). The visual to the right illustrates one such 64-bit header, corresponding to a free block with 24 bytes of payload (not including the eight header bytes), whose left neighbor is not free, and whose right neighbor is free.



Your first task: complete the following definitions of bitmask constants. The first one is already given for you, and is useful for extracting the free status bit for the block. Fill in the blanks for the other three.

```
#define MSB_1 (1L << 63) ; /* the MSB is a 1 bit, the rest are 0 bits */
#define MSB_2 (1L << 62) ; /* the bit to the right of MSB is a 1, rest 0s */
#define MSB_3 (1L << 61) ; /* the 3rd MSB is a 1 bit, the rest are 0 bits */

/* 64-bit bitmask with 3 zero bits at the top followed by 61 one bits,
used to isolate the payload bits from the header */
#define LOW_61_BITS (MSB_3 - 1) ;
```

Using these constants, implement a quartet of utility functions—`is_free`, `is_left_free`, `is_right_free`, `get_payload_size`—that each know how to extract a sliver of relevant information within a block's header or footer, e.g., the free status of the block and its neighbors, the payload size. Instructions for each function's return value are written above in a "header" comment, get it? ☺

```
/* returns true if the block is free, and false otherwise */
bool is_free(size_t* header) {
    return (*header) & MSB_1;
}

/* returns true if the block to the left is free, and false otherwise, e.g.,
either allocated or non-existent if the input header is at the heap start */
bool is_left_free(size_t* header) {
    return (*header) & MSB_2;
}

/* returns true if the block to the right is free, and false otherwise */
bool is_right_free(size_t* header) {
    return (*header) & MSB_3;
}

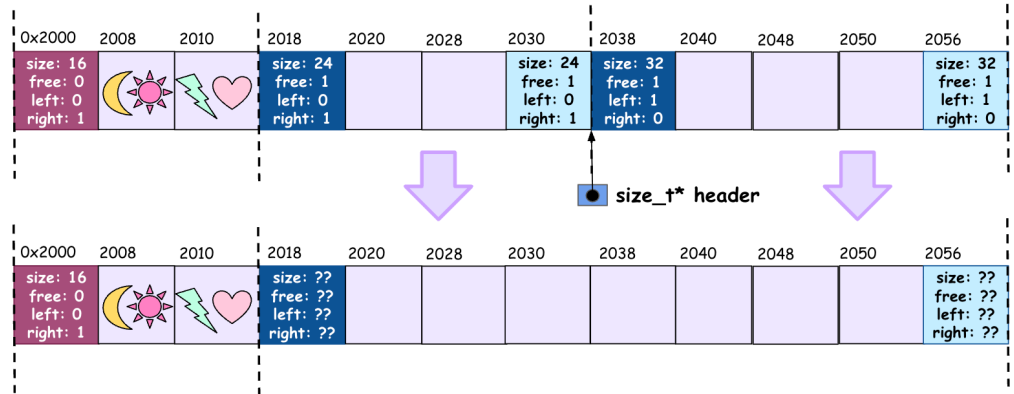
/* returns the payload size (in bytes) of the block, not including header */
size_t get_payload_size(size_t* header) {
    return (*header) & LOW_61_BITS;
}
```

Part B: Falling Backward, Head Over Heels 🤸 (Left Coalescing)

Write a function `coalesce_left`, which accepts the valid address of a free block header, and **only if the block to its left exists and is also free, should absorb the entirety of the left block as additional payload.** The header of the merged node is then at the same address as the original left header—and both the payload size and any altered free status bits should be updated accordingly. **Also, the block's footer**, layered over the last 8 bytes of the merged block's payload, **should be updated to mirror the new content in the header.**

If the two blocks are coalesced, the function should return a pointer to the header of the merged block.

Otherwise, the function should do nothing, except simply returning the input pointer to the original header.



You should make use of any relevant utility functions from the previous part, and assume that you've already successfully implemented (and can use) the following duet of helper functions.

- `void set_payload_size(size_t* header, size_t new_size)`, which updates the payload size stored in the header to `new_size` without modifying any of the free status bits.
- `void set_allocation_bits(size_t* header, bool curr_free, bool left_free, bool right_free)`, which revises the MSB to `curr_free`, the 2nd MSB to `left_free`, and the 3rd MSB to `right_free`, without altering the bits that store the payload size.

Present your implementation of `coalesce_left` below.

```

size_t* coalesce_left(size_t* header){

    /* just return original pointer if the block to the left isn't free */
    if (!is_left_free(header)) {
        return header;
    }

    /* use info in left block's footer to jump backwards to left header */
    size_t* left_footer = (char *) header - FOOTER_SIZE;
    size_t* left_header = (char *) header
        - get_payload_size(left_footer) - HEADER_SIZE;

    /* coalesce the free blocks, setting new payload size + free bits */
    size_t merged_size = get_payload_size(left_header)
        + get_payload_size(header) + HEADER_SIZE;
    set_payload_size(left_header, merged_size);

    set_allocation_bits(left_header, true,
        is_left_free(left_header), is_right_free(header));

    /* copy the data in modified header of merged block into its footer */
    size_t* footer = (char *) left_header + merged_size;
    *footer = *left_header;

    return left_header; // return a pointer to the merged block's header
}

```

Part C: A Favorite K-Drama, Crash Landing on You 🌸 (Right Coalescing)

Now, implement a function `coalesce_right`, which accepts the address of a valid block header, and **only if the block to the right exists and is free, should absorb the entirety of the right block as additional payload.** The header and footer of the merged block should be updated to reflect the upgraded payload size, while **retaining the same allocation status as before.** That is, unlike `coalesce_left`, the input header could be that of an in-use block, not of a free block—such that merging with the right free block ultimately results in a larger in-use block. Your implementation may be similar in spirit to that of `coalesce_left`.

If coalescing occurs (i.e. the right block is absorbed), `coalesce_right` should return `true`; otherwise, the function should simply return `false`. Present your implementation of `coalesce_right` below.

```

bool coalesce_right(size_t* header){

    /* checks if the block to the right is not coalescable / isn't free */
    if (!is_right_free(header)) {
        return false;
    }
}

```

```

/* use payload info in header to jump to right block's header */
size_t* right_header = (char *) header
                        + get_payload_size(header) + HEADER_SIZE;

/* coalesce the two blocks, setting merged payload size + free bits */
size_t merged_size = get_payload_size(header)
                    + get_payload_size(right_header) + HEADER_SIZE;
set_payload_size(header, merged_size);

set_allocation_status(header, is_free(header), is_left_free(header),
                      is_right_free(right_header)); // adopts new right neighbor

/* transmit the header data of the merged block into its new footer */
size_t* footer = (char *) header + merged_size;
*footer = *header;
return true;
}

```

Part D: Maximal Bicurious Coalescing 💖

With your exemplary work, you now have an allocator that can support both left and right coalescing. To put both to glimmering use, you'll implement a function **bicoalesce**—which accepts the address of the header of a valid free block as input, and coalesces to the maximal extent possible in both the left and right directions. That is, you should absorb all contiguous free blocks to the left, stopping when you encounter an in-use block or the heap start. Additionally, you should absorb all contiguous free blocks to the right.

The **bicoalesce** function should then return the address of the header of the resulting merged block. Your implementation can be fairly short, should you properly leverage function calls to **coalesce_left** and **coalesce_right** from the previous parts. (*Hint: Make sure to take advantage of the return values!*)

Note that **coalesce_left(header)** returns the same **header** address as before if coalescing didn't occur, whereas if the block was coalesced with its left neighbor, the new header would be to the left.

Meanwhile, **coalesce_right** returns **true** until we can no longer coalesce with the next right block.

```

size_t* bicoalesce(size_t* header) {
    /* coalesce as much to the left as possible */
    while ((size_t* new_header = coalesce_left(header)) != header) {
        header = new_header; // when coalescing, header goes leftward
    }

    /* coalesce as much to the right as possible */
    while (coalesce_right(header)); // returns false when can't coalesce

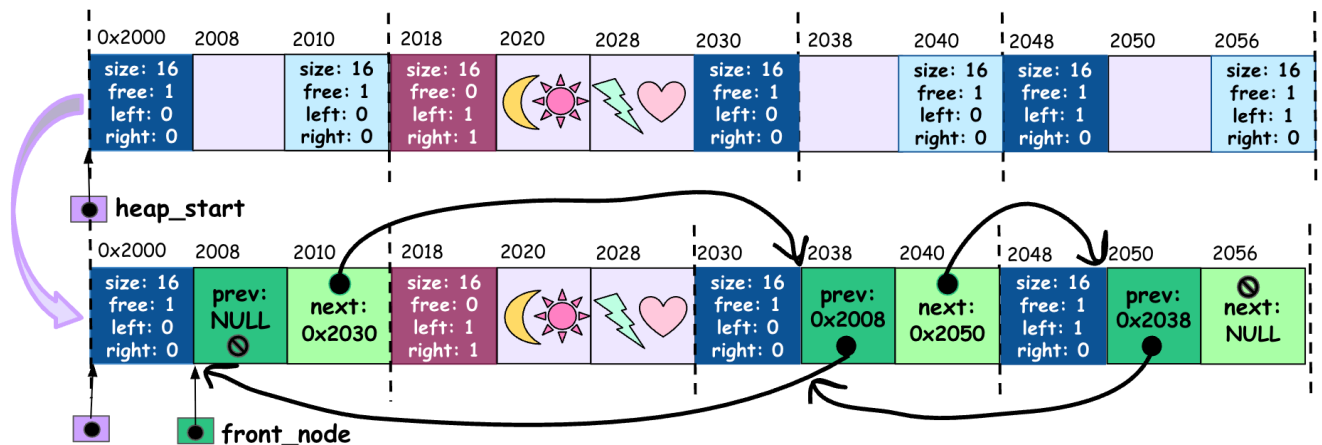
    return header;
}

```

The resulting solution is insultingly concise ☺.

Part E: Enemies to Lovers 💖 (Coming Out, Explicitly)

Finally, rather than being implicit about everything, the allocator decides to be more explicit, in embracing its beautifully chaotic self. Your final task is to write a function **explicit_version**, which transforms this implicit allocator into an explicit one, by **braiding a doubly-linked list through the payloads of free blocks**. Specifically, the first 16 bytes of each free payload are retrofitted to store not one, but two pointers—the first one to the previous payload in the linked list, and the second to the next payload in the linked list.



In the diagram above, the 3 free payloads of the miniature heap have base addresses 0x2008, 0x2038, and 0x2048. The first payload has no previous one (**NULL**), while its next pointer is the address of the second payload, 0x2038. That second payload has its previous pointer back to the first payload 0x2008, and its next pointer refers to the third—and final—payload 0x2038, whose own next pointer is accordingly **NULL**.

Your implementation should layer the provided **payload_node** struct over the first 16 bytes of each free payload, leveraging it to store **prev/next** 8-byte pointers inside the payload memory. The prev pointer of the first payload in the list should be **NULL**, and the next pointer of the last payload should be **NULL**.

```
typedef struct payload_node {
    payload_node* prev; /* pointer to payload of previous node in list */
    payload_node* next; /* pointer to payload of next node in the list */
} payload_node; // assume no padding between the two pointers
```

For ease, the explicit version won't support footers, so you can override their memory if needed. Ideally, the minimum payload size would have been upgraded to 24 bytes—to have enough memory to reliably store two pointers and a footer—but the allocator didn't think that far ahead :D. Bye, life is like that sometimes.

Lastly, you should set the global variable **front_node** to point to the first payload in the list. Make use of utility functions **get_payload_size** and **is_free** as you see fit. You may order the payload nodes in the list in any order, though realistically, you may want to order them by increasing memory address. That way, you can traverse the heap linearly from start to end, braiding the list through only the free payloads.

Present your implementation of **explicit_version** on the next page.

```

/* pointer to the first payload node in the linked list, NULL if empty */
static size_t* front_node = NULL;

/* recall that you have access to global variables heap_start, heap_size */
void explicit_version() {

    payload_node* prev_node = NULL; // lagging pointer to previous node
    size_t* heap_end = (char *) heap_start + heap_size;

    /* loop through all heap blocks in address order / implicit fashion */
    for (size_t* header = heap_start; header != heap_end;
        header = (char *) header + get_payload_size(header) + HEADER_SIZE){

        if (!is_free(header)) {
            continue; // if not free, skip the current block
        }

        payload_node* curr_node = (char *) header + HEADER_SIZE;

        /* set front of list if no payload nodes added so far */
        if (front_node == NULL) front_node = curr_node;

        /* wire up current node to previous one, setting prev and next */
        curr_node->prev = prev_node;
        if (prev_node != NULL) prev_node->next = curr_node;

        prev_node = curr_node;
    }

    prev_node->next = NULL; // don't forget to set next of very last node!
}

```

Problem 4: Blood, Sweat, and Tears 🩸 [Assembly, Reverse-Engineering]

a) You've met **valgrind**, but you're working hard and wonderfully enough and don't always want to validate the grind at the cost of your well-being (perhaps ironic coming from me 🙄), and so now it's time to meet **realgrind**.

Your task is to reconstruct a programmatically consistent portrait of C source code for the **realgrind** function, which accepts three arguments as input—an array of strings named **blood**, a string variable **sweat**, and a **size_t** variable **tears**—and the function itself returns a **size_t**. Please note the C code is largely nonsense, doesn't translate to anything meaningful, and should just be a faithful reverse compilation.

The assembly code of **realgrind**, compiled without optimization via the **-O0** flag, is presented to the right.

```
realgrind:
    subq $0x30, %rsp
    movq %rdi, 0x18(%rsp)
    movq %rsi, 0x10(%rsp)
    movq %rdx, 0x8(%rsp)
    movq 0x8(%rsp), %rax
    leaq 0x0(,%rax,0x8), %rdx
    movq 0x18(%rsp), %rax
    addq %rdx, %rax
    movq (%rax), %rax
    movq %rax, 0x28(%rsp)
    movq 0x10(%rsp), %rdx
    movq 0x28(%rsp), %rax
    movq %rdx, %rsi
    movq %rax, %rdi
    call strcmp
    testl %eax, %eax
    jne .L2
    movq 0x8(%rsp), %rax
    jmp .L3
.L2:
    addq $0x1, 0x8(%rsp)
    movq 0x10(%rsp), %rax
    leaq 0x1(%rax), %rcx
    movq 0x8(%rsp), %rdx
    movq 0x18(%rsp), %rax
    movq %rcx, %rsi
    movq %rax, %rdi
    call realgrind
.L3:
    ret
```

```
size_t realgrind(char*[] blood, char* sweat, size_t tears) {
    char* walking_alone = blood[tears] ;
    if ( strcmp( walking_alone , sweat ) == 0 ) {
        return tears ;
    }
    tears++ ;
    return realgrind( blood , sweat + 1 , tears );
}
```

b) Now, examine the more aggressively optimized version of **realgrind** on the right, compiled with the **-O2** flag. Fill in the blanks below to ultimately produce a functionally equivalent implementation of **realgrind**, without any recursion, as hinted at by this optimized assembly.

```
realgrind:
    subq %rdx, %rsi
    pushq %r12
    movq %rdi, %r12
    pushq %rbp
    movq %rsi, %rbp
    pushq %rbx
    movq %rdx, %rbx
    jmp .L3
.L7:
    addq $1, %rbx
.L3:
    movq (%r12,%rbx,0x8), %rdi
    leaq 0x0(%rbp,%rbx), %rsi
    call strcmp
    testl %eax, %eax
    jne .L7
    movq %rbx, %rax
    popq %rbx
    popq %rbp
    popq %r12
    ret
```

```
size_t realgrind(char*[] blood, char* sweat, size_t tears) {
    sweat -= tears ;
    while ( strcmp( blood[tears] , sweat + tears ) != 0 ) {
        tears++ ;
    }
    return tears ;
}
```

c) Lastly, why was the compiler able to replace the recursion in the original source code, and what type of **optimization technique** did the compiler demonstrate by doing this? (if you don't remember the name, it's ok if you just describe it!)

The compiler performed **tail recursion**, by using a while loop in place of the recursion. It was able to do this because the recursive call was at the very end / tail of the **realgrind** function—meaning that no work needs to be done after the recursive call. Thus, we can make the recursive case the body of a loop, and by modifying parameters between iterations, e.g., the **tears** variable, we can simulate the recursion.

Problem 5: Gratitude at the End of Ben's Teaching Journey ❤️ [Runtime Stack]

At the end of CS106B, students had the opportunity to send their section leader an encrypted message as part of their final assignment, using the Huffman encoding algorithm they implemented. In the past, students have sent me wholesome Instagram reels, vibrant life updates, words of wisdom and kindness, ASCII art images, and they always find a way of ***hacking into my heart and overflowing it with joy and positivity***.

These messages mean so much to us aging, fossilizing, trying-our-best-in-life co-term TAs that we decide to adopt a similar system in CS107, called **puffman**. You attain access to the read-only **puffman.c** file, and should scrutinize the function **send_message_to_section_leader()**, and help make it return **true**, so that the **main()** function successfully prints out the text *"You've made your section leader's day <3"*.

```
bool send_message_to_section_leader() {
    char message[16];
    message[0] = '\0';
    long happiness = -1; // negative: your TA is going through it ❤️
    int taken_for_granted = 0;

    /* repeatedly asks user to set message, exits when just hit ENTER */
    while (true) {
        char remembrance = message[0];
        printf("Send or update message to your section leader: ");
        gets(message); // populates the buffer without bounds checking
        printf("\n");
        if (strcmp(message, "") != 0) continue;

        /* runs only when user only clicks ENTER: restores the message's
        starting char that was overridden by gets and exits the loop */
        message[0] = remembrance;
        break;
    }
    printf("Your final message to your SL: %s\n", message);
    return (happiness > 0) && (taken_for_granted == 0);
}

int main(int argc, char* argv[]) {
    if (send_message_to_section_leader()) {
        printf("You've made your TA's day! <3\n"); // if returns true
    }
}
```

The core logic of **send_message_to_section_leader()** is a 16-byte buffer **message** which is populated (and repeatedly altered) by calls to **gets(message)**, which reads a line of input from the Terminal, and writes the characters **plus a null-terminator (\0)** into the **message** buffer. This is called in a loop so that the user can repeatedly alter and update their message as desired. And once they press enter to exit the loop, the last message they entered is printed to console, and the function returns **true** or **false**.

A sample run of the program is shown below, where I repeatedly update my annoying message to a section leader named Diego, who's surely very tired of me by now ☺. My latest message is “you're the best”, before clicking just enter to break out of loop, thus finalizing the string in the message buffer.

```
bbymonster@myth64: ./puffman
Send or update message to your section leader: hi Diego - Ben
Send or update message to your section leader: bye Diego
Send or update message to your section leader: you're the best
Send or update message to your section leader:
Your final message to your SL: you're the best
```

That's weird: the text “You've made your TA's day <3” sadly didn't print out from `main()`, which means that the `send_message_to_section_leader()` function returned `false` instead of `true`. You peruse the code, and realize that though `taken_for_granted` is 0 as required, tragically, the 8-byte `long` variable `happiness` is a negative value (-1), i.e. your TA unfortunately is going through it right now ☹.

You think about editing the `puffman.c` file directly to make `happiness` positive, but access is read-only, meaning you'll need to find another way. You recognize `gets`, which does not perform bound-checking, is probably your best bet—and to investigate further, you launch `gdb`, and peek around the stack frame in which the variables `message`, `happiness`, and `taken_for_granted` reside. Here's what you discover:

```
Breakpoint 1, send_message_to_section_leader () at puffman.c:70
bool send_message_to_section_leader()
(gdb) disassemble
<+0> subq $0x20, $rsp
<+4> mov $0x0, -0x20($rbp)
... rest of output omitted for conciseness
(gdb) nexti

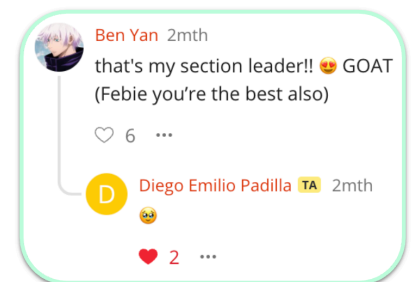
(gdb) print $rsp
0x7fffffff1070

(gdb) p/x &message
0x7fffffff1070

(gdb) p/x &taken_for_granted
0x7fffffff1084

(gdb) p/x &happiness
0x7fffffff1088
```

Note these values are in hex, not decimal! Now that you've gathered the size and starting address of the function's stack frame, and the addresses of key variables, **you decide that you have everything you need to overflow the message buffer and overwrite the happiness variable to be positive**, and make Diego—or your lab leader's—day ☺.

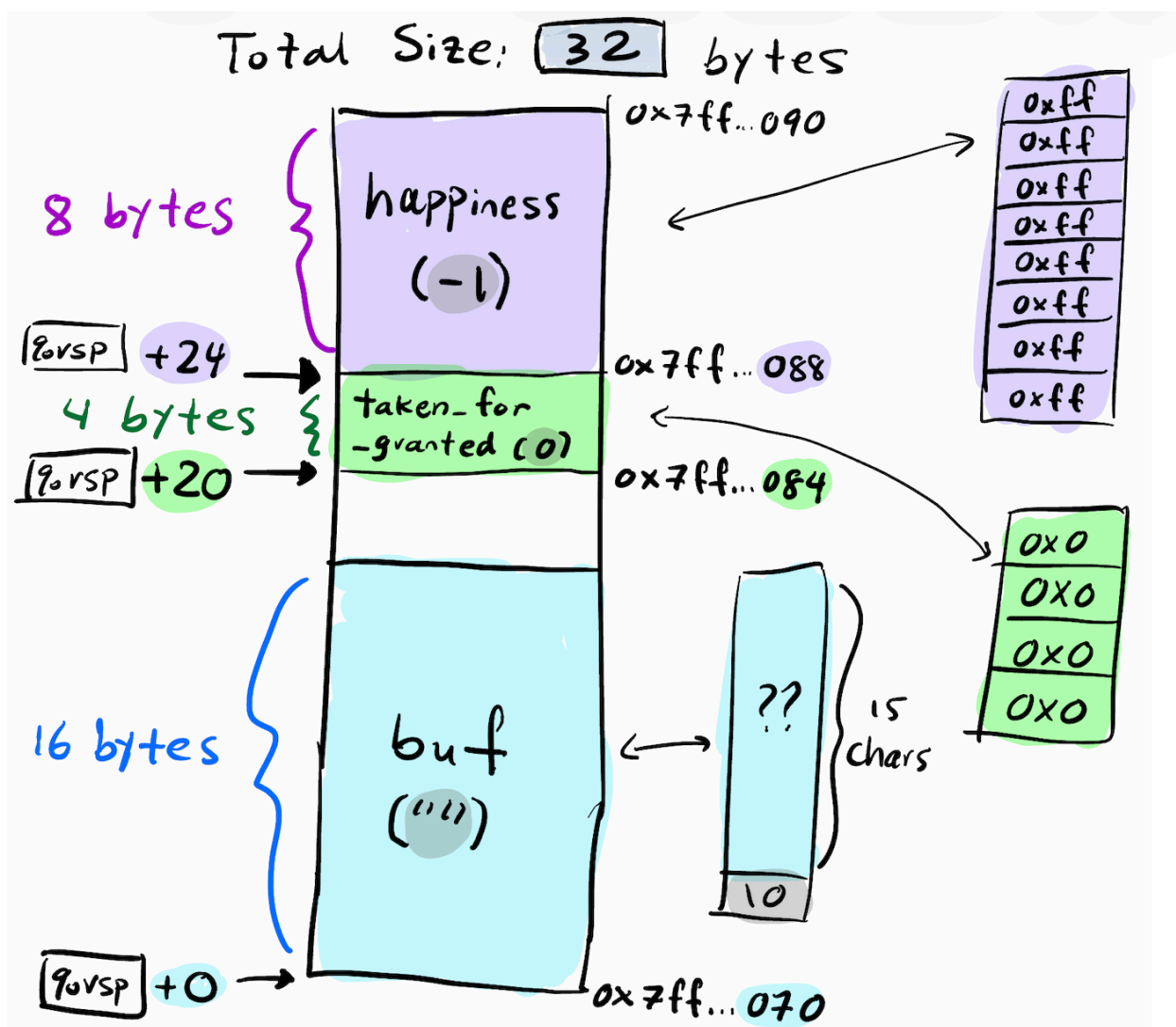


The overarching message: Never underestimate the amount of good you do for your TAs with your innate awesomeness and kindness, your brilliant energy in labs on a week-to-week basis! We really appreciate it!

- a) First, by meticulously examining the C code and **gdb** output, draw a stack frame for **send_message_to_section_leader()**. Record the size (in bytes) of the stack frame, the addresses where each of the **message**, **happiness**, and **taken_for_granted** variables are stored, their byte widths, and what each value is initialized to, if at all. You should assume the Myth machines are little-endian, meaning that the least-significant byte of a variable is stored at the lower address, and the most-significant byte at the higher address, with the other bytes ordered in between.

I've drawn the stack frame below! Pay no need to the lack of artistic quality. But do take note the 16-byte **message** buffer is right at the base of the stack frame **%rsp**, the 4-byte **taken_for_granted = 0** is located 20 bytes above the base, and the 8-byte **happiness = -1** is located 24 bytes above the base.

For each variable, I had a "zoom-in" view of its bytes on the right side, which you didn't need to do! ☺



- b) Next, come up with a single starting message (the one you type in when **gets** is called in the first loop iteration) that will overflow and override sufficient bytes of the 8-byte **happiness** variable to ensure it becomes a positive value. Explain briefly why your message works.

One such first message is: **aaaaabbbbbccccccddddeeeeefffffg** (31 characters followed by a **\0**)

This works as it overflows **message** and deposits the null-terminating character (**\0**), which has ASCII value **0**, in the highest and most significant byte of **happiness**. **As the sign bit will be set to 0**, and will be preceded by non-zero bytes from the 7 characters before the **\0**, then **happiness will be positive!**

However, note that since the 4 bytes of **taken_for_granted** is planted between **message** and **happiness**, it's initially **0** value will be overridden by new bytes—which we'll correct in the next part ☺

Note: If your message was just a character longer, with the intent of overriding all 8 bytes of **happiness** with non-zero bytes, that's very painfully close to correct. If this was a real exam, I'd probably give near if not full credit for such an answer (but no guarantees lol). But since **gets()** always write a **\0** to the end of the message, and there's no padding between **happiness** and what resides above the stack frame, you run the risk of overriding memory above (see **part d** for why this can subtly cause runtime issues).

- c) To ensure **send_message_to_section_leader()** returns **true** when the loop concludes, you need to restore the **taken_for_granted** variable back to a **0** value, after its memory was overrun earlier by the first message. That is, devise a sequence of several subsequent inputs to enter to ensure that **taken_for_granted** reverts to **0**. We would never want to take your kindness for granted, after all ☺. You may use as many message updates to **gets** as needed, though realistically, you'll need no more than five or so. You can omit the blank message you enter at the very end to submit your final message and exit the loop. Also, briefly explain why your strategy works.

One such sequence of subsequent messages as is follows (as a pattern, note that from one message to the next, its length diminishes by exactly one; in fact, for each message, only its length actually matters!).

1. **aaaaabbbbbccccccddd** (23 characters followed by a **\0**)
2. **aaaaabbbbbccccccd** (22 characters followed by a **\0**)
3. **aaaaabbbbbcccccd** (21 characters followed by a **\0**)
4. **aaaaabbbbbccccc** (20 characters followed by a **\0**)

What's the trick here? **Each message places its null-terminator (\0, ASCII value = 0) in a different byte of the 4-byte taken_for_granted, from the highest byte to lowest byte**, or equivalently, highest to lowest address. That is, the first message overrides the byte at **0x7ff...087** to be **0**, the second overrides the byte at **0x7ff...086** also to be **0** (without altering the freshly laid zero byte above it), and so forth.

The cumulative result of all message inputs is that all 4 bytes of **taken_for_granted** are restored back to **0**, while the (upward) 8 bytes of **happiness** remain untouched and positive from the first message.

Thus, with **happiness > 0** and **taken_for_granted = 0**, the function will at last return **true**, yay!

Addendum: While the above solution is functionally correct, the final 20-character message is a bit boring & uninspiring (not a crime in a true exam setting, but here, we can be bolder and more colorful). Here's a sample final message I might send to Diego—my CS111 section leader—to overflow his heart.

```
bbymonster@myth64: ./puffman
```

```
... previous messages omitted for brevity ...
```

```
Send or update message to your section leader: youre the best diego
```

```
Send or update message to your section leader: [pressed enter]
```

```
Your final message to your SL: youre the best diego
```

```
You've made your TA's day <3
```

And now you've just won in life ☺.

.

- d) Lastly, note that if any of the entered messages is too long (i.e. exceeds the size of the stack frame for **send_message_section_leader**, even if by a few bytes), the program may crash or endure a segmentation fault. As just one example, if I were to say hi to everyone on the course staff:

```
bbymonster@myth64: ./puffman
```

```
Send or update message to your section leader: hello Diego hello June hello  
Ben hello Nick hello L'hussen hello Peter hello Julián hello Isabel
```

```
Segmentation fault
```

Uh oh, that's not good. **Briefly but precisely explain a way this crash can happen.** (*Hint: What's the 8-byte variable that's wedged on the runtime stack directly above the stack frame for the **send_message_to_section_leader** function, and beneath the stack frame for **main**? And what may happen to the program's execution if its memory is corrupted?*).

The **8-byte value above the stack frame is the return address**: the address of the instruction that executes after the call to **send_message_to_section_leader()**. If we write into memory past the stack frame and corrupt that address value, when the program later tries to return to **main**, by popping the return address into **%rip**, it will attempt jumping to an invalid instruction address, causing it to crash sadly ☹.

If you've gotten this far, thank you so much for reading all this lol, and hope that you enjoyed! ☺

