CS107 December 11th, 2025

Autumn 2025

# CS107 Final Examination

This is a closed book, closed note, closed computer exam, though you're permitted to refer to the reference sheet I've provided.

You have 180 minutes to complete all problems. You don't need to **`#include`** any libraries, and you needn't use **`assert`** to guard against any errors. Understand that most points are awarded for concepts taught in CS107, and not prior classes. You don't get many points for **`for`**-loop syntax, but you certainly get points for proper use of **`&`**, **`*`**, and the low-level C functions introduced in the course. If you're taking the exam remotely and have questions, you can text or telephone Jerry at 415-205-2242.

Good luck!

SUNet ID (@stanford.edu): _____

Full Name: _____

I accept the letter and spirit of Stanford's Honor Code.

[signed] _____

| | | Score | Grader |
|---|---|---|---|
| 1. Scheme and List Serializations | [10] | _____ | _____ |
| 2. x86-64 and **`gcc`** optimizations | [20] | _____ | _____ |
| 3. Runtime Stack | [20] | _____ | _____ |
| 4. Allocators | [40] | _____ | _____ |
| **Total** | **[90]** | _____ | _____ |

**Problem 1: Scheme and List Serializations [10 points]**

Scheme is a programming language whose primary data structure is **the linked list**. Unlike any of the linked lists you've dealt with in C, Scheme lists are fully heterogeneous—that is, the entries needn't all be the same type.

Some example lists are:

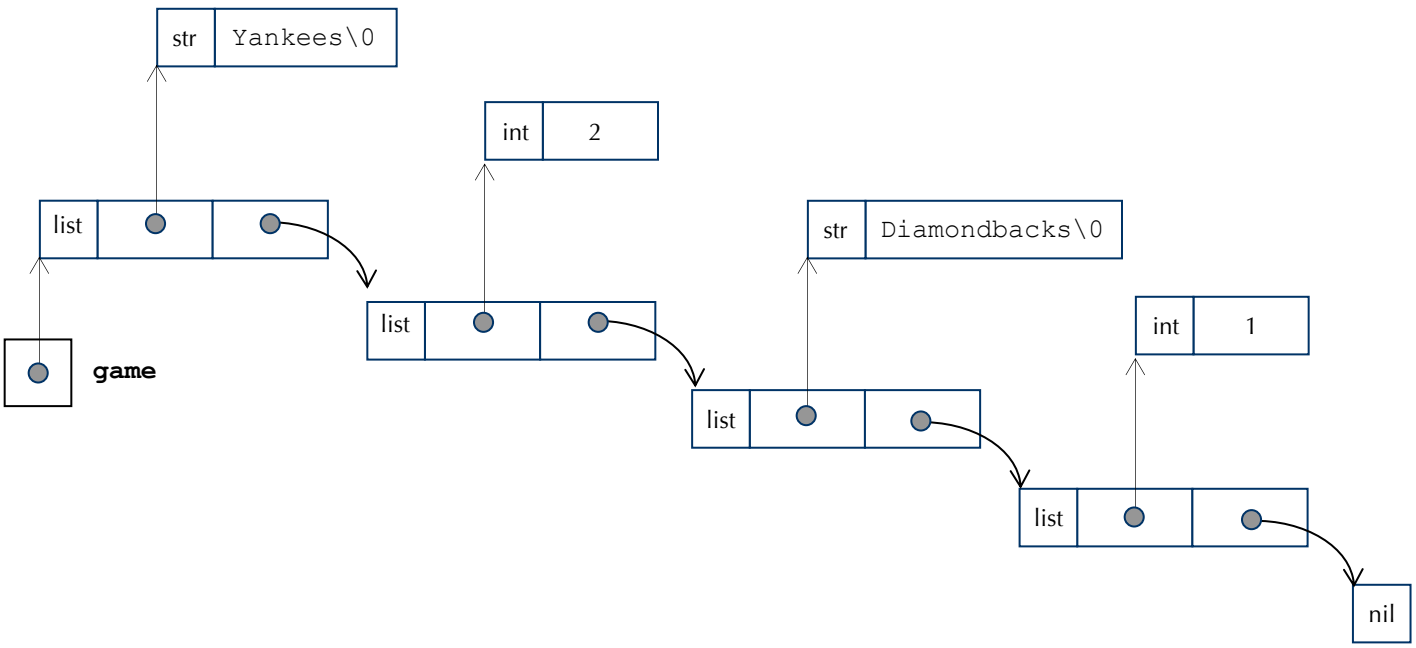
- `(2 3 5 7)`
- `(House at Pooh Corner)`
- `(Yankees 2 Diamondbacks 1)`
- `(4 calling birds 3 French hens 2 turtle doves 1 partridge)`

Scheme's linked lists are so versatile that individual elements might themselves be lists. And that being the case, lists can be nested to any depth.

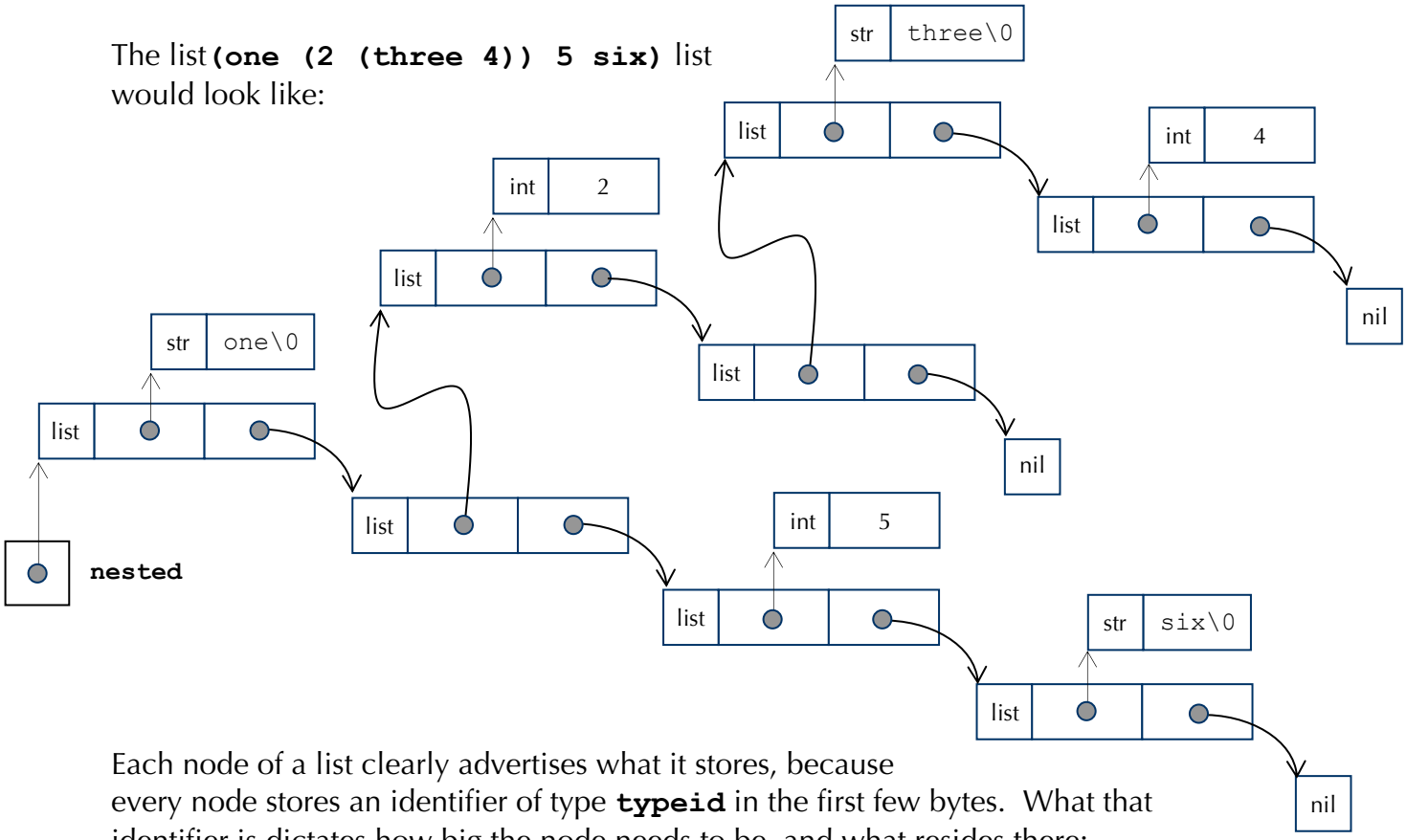
- `((1 2) (buckle my shoe))`
- `(one (2 (three 4)) 5 six)`
- `(how (nested (can (u (go)))) how (nested (can (u (go)))))`

We can support heterogeneous lists in C, but they don't come easy. In order for them to work, individual elements of the list must **carry their type information**. The idea is to tag each list node with some enumerated type that tells us what the rest of the node contains.

We'll pretend that integers and strings are the only atomic types of interest. The third list above (if bound to the stack variable **`game`**) would be structured as follows:



The list **`(one (2 (three 4)) 5 six)`** list would look like:

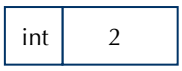



Each node of a list clearly advertises what it stores, because every node stores an identifier of type **`typeid`** in the first few bytes. What that identifier is dictates how big the node needs to be, and what resides there:

We'll officially allow strings and integers; therefore, the following enumerated type suits our needs:

```
typedef enum {
   Integer = 0, String = 1, List = 2, Nil = 3
} typeid; // typeid vars can equal Integer, String, List, or Nil
```

When handed a list, we extract the **`typeid`** value from the first **`sizeof(typeid)`** bytes. From that, we know whether the rest of the node…

stores an **`int`**, as with: 


stores a null-terminated character array, as with: 


stores two addresses, as with: 

, or

stores nothing—the end of some list has been reached. 


Note the characters of a **`String`** reside directly within the node.

The **serialize** function takes a well-formed list and returns the ordered concatenation of all of the list's strings and integers as one, big dynamically allocated string, including those in nested sub-lists. You can assume you have access to a **int_to_string** function, which accepts an **int** and returns a dynamically allocated C string, e.g., **int_to_string(107)** returns a heap-based **"107"**. Your implementation should not orphan any memory while constructing the serialization, though you shouldn't free any memory associated with the original list.

```
/**
 * Traverses a properly structured list, and returns the ordered
 * concatenation of all strings, including those in nested sub-lists.
 * When applied to the two lists drawn above, the following strings
 * would be returned:
 *
 *    serialize(game) would return "Yankees2Diamondbacks1"
 *    serialize(nested) would return "one2three45six"
 */

typedef enum {
   Integer = 0, String = 1, List = 2, Nil = 3
} typeid; // typeid vars can equal Integer, String, List, or Nil

char *serialize(typeid *list) {
   assert(*list >= Integer && *list <= Nil);

   if (*list == Nil) return strdup("");
   if (*list == Integer) return int_to_string(*(int *)(list + 1));
   if (*list == String) return strdup((char *)(list + 1));

   typeid **pointers = (typeid **)(list + 1);
   char *front = serialize(pointers[0]);
   char *back = serialize(pointers[1]);

   char *combined = realloc(front, strlen(front) + strlen(back) + 1);
   strcat(combined, back);
   free(back);
   return combined;
} // implementation can also call malloc, strcpy and strcat
  // in front and back, and free front and back
```

**Problem 2: x86-64 and gcc Optimizations [20 points]**

Below are two versions of the assembly code generated for the **doris** function. The version on the left was compiled **-O0**, and the right was compiled **-O2**.

```
doris:
    sub $0x28,%rsp
    mov %rdi,0x8(%rsp)
    mov %rsi,(%rsp)
    mov (%rsp),%rax
    shl $0x3,%rax
    mov %rax,0x18(%rsp)
    mov 0x8(%rsp),%rax
    mov (%rax),%rax
    mov %rax,0x10(%rsp)
    jmp .L3
.L1
    lea 0xe7b(%rip),%rax
    mov %rax,%rdi
    call <strlen>
    mov %rax,%rdx
    mov 0x10(%rsp),%rax
    cmp %rax,%rdx
    jne .L2
    mov $0x0,%eax
    jmp .L4
.L2
    mov 0x18(%rsp),%rdx
    mov %rdx,%rax
    add %rax,%rax
    add %rdx,%rax
    add $0x1,%rax
    add %rax,0x18(%rsp)
    subq $0x7,0x10(%rsp)
.L3
    call <random>
    cmp %rax,0x10(%rsp)
    jne .L1
    mov 0x18(%rsp),%rax
    lea 0x0(,%rax,8),%rdx
    lea 0x8(%rsp),%rax
    add %rdx,%rax
    mov (%rax),%rax
.L4
    add $0x28,%rsp
    ret
```

```
doris:
    push %rbp
    lea 0x0(,%rsi,8),%rbp
    push %rbx
    sub $0x18,%rsp
    mov (%rdi),%rbx
    mov %rdi,0x8(%rsp)
    jmp .L2
.L1
    cmp $0x5,%rbx
    je .L3
    lea 0x1(,%rbp,4),%rbp
    sub $0x7,%rbx
.L2
    call <random>
    cmp %rbx,%rax
    jne .L1
    mov 0x8(%rsp,%rbp,8),%rax
    add $0x18,%rsp
    pop %rbx
    pop %rbp
    ret
.L3
    add $0x18,%rsp
    xor %eax,%eax
    pop %rbx
    pop %rbp
    ret
```

a) [14 points] Fill in the blanks below so that the implementation of **doris** matched the **unoptimized** assembly on the prior page. Note that this is nonsense code and isn't intended to do anything meaningful. You're simply working to faithfully translate assembly back to C. (You may assume the only string constant in the entire implementation is **"CS107"**.)

```
void *doris(void *monkey, long dragon) {
    long lambie = 8 * dragon;
    for (long ball = *(long *)monkey; ball != random(); ball -= 7) {
      if (ball == strlen("CS107")) return NULL;
      lambie += 3 * lambie + 1; // or lambie = 4 * lambie + 1
    }
    return (&monkey)[lambie];
}
```

b) [2 points] The string constant of interest is presumably identified as **0xe7b(%rip)**, since it's the first operand of the **lea** instruction directly preceding the **strlen** call. Why can the string constant be expressed as an **%rip**-relative address?

> At the time that instruction is executed, **%rip** is the address of the following instruction, which is a known constant. The base address of the 'C' in CS107 must be equal to **%rip + 0xe7b**. (The fact that string constants are close in memory to assembly instructions shouldn't surprise you—the data segment is right on top of the code segment in memory.)

c) [2 points] The unoptimized code has a call to **strlen**, whereas the optimized version does not. Where did it go?

> The compiler trusts that the **strlen** call is that of a well-known built-in, since it calls it during compilation instead of allowing it to be called at runtime. This is a form of constant folding (though you didn't need to refer to it by that name.)

d) [2 points] The unoptimized version includes four **add** instructions between the **.L2** and **.L3** labels, whereas only the last of the four **add**s is retained in the optimized version. Which instruction in the optimized version replaces the first three **add**s?

> These add instructions are in place to compute **lambie += 3 * lambie + 1**, Because lambie is going up a factor a 4, the brute-force approach of multiple adds can be replace by an instruction that multiplies by 4 and adds one all in one breath. That instruction is **lea 0x1(,%rbp,4),%rbp**.

**Solution 3: Runtime Stack [20 points]**

The famously sloppy bank programmer is at it again. His latest buggy addition is the **authorize** function shown below (C source on left and generated assembly on the right). (During compilation, there is a warning about a missing return statement.)

```
struct customer {
   char username[16];
   long balance;
   int high_security;
   int priority;
};

struct database {
   int count; // num customers
   char bankname[20];
   struct customer array[500];
};

bool authorize(long id, struct database *db) {
   if (id >= 0 && id < db->count)
      return (strcmp(getenv("USER"),
              db->array[id].username) == 0);
}
```

```
authorize:
    test   %rdi,%rdi
    js     .L1
    movslq (%rsi),%rax
    cmp    %rax,%rdi
    jge    .L1
    push   %rbx
    shl    $0x5,%rdi
    lea    0x18(%rsi,%rdi,1),%rbx
    mov    $0x4019c7,%edi
    callq  <getenv>
    mov    %rbx,%rsi
    mov    %rax,%rdi
    callq  <strcmp>
    test   %eax,%eax
    sete   %al
    movzbl %al,%eax
    pop    %rbx
.L1
    retq
```

a) [3 points] Why does the generated assembly use the caller-owned register **%rbx** when there are plenty of callee-owned registers available?

> The result placed in **%rbx** is more easily computed before the call to **getenv** (because it depends on the pre-**getenv** value of **%rdi**), but it isn't used until *after* the **getenv** call. The compiler could have made this work using callee-owned registers, but it biases toward caller-owned registers when partial results need to be maintained across function calls.

b) [4 points, 2 and 2] Explain how each of the constants—**0x05** and **0x18**—in these two instructions is determined:

```
shl $0x5,%rdi
lea 0x18(%rsi,%rdi,1),%rbx
```

> Each **struct customer** is 32 bytes in size. **%rdi**, which stored the array offset **id**, needs to be scaled by 32 bytes to find the address of the **id**th element in **db->array**. Multiplying by 32 is the same as left shifting by **0x5**.
>
> The **0x18** is really the number 24, and 24 is the offset of the **struct customer** array within the **struct database** referenced by **db**.

The **authorize** function purports to look up the account for the supplied **id** and confirm the account name matches the logged in user's name. The function returns the right thing for a valid **id**, but it may or may not behave well when supplied with a bogus (though still well-formed) **id**. Assume **db** addresses a well-formed database for each of the calls below.

c) [3 points] Consider the code snippet below, and comment on whether or not the call to **authorize** will return **true**, return **false**, or crash. If more than one thing might happen, then say so. Regardless of the possibilities, defend your answer.

```
if (authorize(-1, db))
    printf("I'm in!\n");
```

When **id** is -1, **authorize** doesn't touch **%rax**. The call certainly won't crash, but the return value will be **false** if **%rax** incidentally contained a 0 at the time **authorize** was called, but otherwise it'll return **true**.

d) [3 points] Now consider a second code snippet, and comment on whether or not the call to **authorize** will return **true**, return **false**, or crash. If more than one thing might happen, then say so. Regardless of the possibilities, defend your answer.

```
if (authorize(LONG_MAX, db))
    printf("$$$$$$$$$$\n");
```

If you look at the C code, you can tell that no matter what **db->count** is, it's not going to be strictly larger than **LONG_MAX**. That means the function will exit without executing the return statement, the value in **%rax** will be, as always, taken to be the return value.

What's in **%rax**, you ask? A sign-extended copy of **db->count**! So, if it's 0, then **authorize** returns **false**. Otherwise, it returns **true**.

The **confirm_passcode** function below emulates one of the many vulnerabilities in the **atm** problem from **assign5**.

```
// this string contains 16 chars
#define DIR "atm/bankrecords/"

bool confirm_passcode(const char *name,
                      int passcode) {
    int cust_code = 0;
    char path[sizeof(DIR) + sizeof(name)];

    sprintf(path, "%s%s", DIR, name);
    return cust_code != 0 &&
           cust_code == passcode;
}
```

```
confirm_passcode:
    push   %rbx
    sub    $0x20,%rsp
    mov    %rdi,%rcx
    mov    %esi,%ebx
    movl   $0x0,0x1c(%rsp)
    mov    $0x4019cf,%edx
    mov    $0x4019e2,%esi
    mov    %rsp,%rdi
    mov    $0x0,%eax
    callq  <sprintf>
    mov    0x1c(%rsp),%eax
    test   %eax,%eax
    je     .L1
    cmp    %ebx,%eax
    je     .L2
.L1:
    // truncated in answer key
```

You predict that breaking into the vault will be as simple as setting up a call to

**confirm_passcode("BANKVAULT7777777777777777", 0x7777).**

However, the above call crashes. You investigate using **gdb** and get this:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000037373737 in ?? ()
```

e) [7 points, 3 and 4] Explain the problem and then present the correct call that will fool **confirm_password** into returning **true**.

First, explain why the call crashes.

> The supplied name is so long that it overruns path and overwrites the saved return address on the stack. When **confirm_passcode** tries to return, it loads %rip with an invalid instruction address and crashes.

Next, specify a pair of arguments that'll coerce **confirm_password** to return **true**.

**confirm_passcode("BANKVAULT777<u>7</u>", '7');**

> This version writes two bytes into **cust_code**, the digit character **'7'** and the terminating **'\0'**. The passcode is the ASCII value of digit character **'7'** could instead by written as **0x37** (evident from the crash address reported by **gdb**) or **55**. If student didn't know about endianness, then **'7' << 24** is acceptable as well.

**Solution 4: Allocators [40 points]**

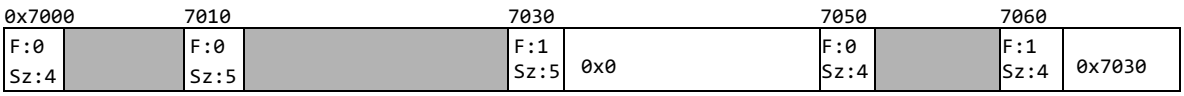
You are writing code for a simple allocator that uses a block header, no footer, and maintains an explicit free list. Implementation details of this allocator include:

- The total block size—that is, the size of the payload plus the size of header—for all blocks is required to be a power of 2. And since the payload must be at least 8 bytes and headers are 1 byte, the minimum total block size is 16. Payload pointers are not required to be aligned.

- The block header is an 8-bit **`unsigned char`** that bit-mashes together the block information. The most significant bit is 1 if the block is free, and it's 0 otherwise. The remaining bits store the $\log_2$ of the total block size. So, if a block size is 16, then these seven bits would encode a 4. If a block size is 32, these seven bits would store a 5.

- The allocator maintains an explicit free list as a singly linked list. A global variable points to the header of the first free block (or **`NULL`** if the free list is empty). The first 8 bytes of the payload of a free block store a pointer to the header of another free block. The last free block on the list stores **`NULL`** in its payload.

Here is what a small heap might look like after a few requests have been serviced.




The above heap starts at address 0x7000 and has size 112 bytes. Three blocks are in use, two blocks are free, and the allocated payloads are shown shaded in gray. The header of the first block has the most significant bit off (block is in-use) and the lower bits store a 4 (total block size is 16, or $2^4$). The free list points to the header at 0x7060, the payload of that block points to the header at 0x7030, and the payload of that block stores a **`NULL`**.

Here are our allocator's global variables, constants, and **`typedef`**s.

```
// type: header stores per-block housekeeping
typedef unsigned char Header;

static Header *heap_start; // first header in heap segment
static size_t  heap_size;  // number of bytes in heap segment
static Header *free_list;  // header of first free block (or NULL)

// masks to divide header into most significant bit and the others
#define MSB_MASK   (1 << ((sizeof(Header) << 3) - 1))
#define SIZE_MASK  (MSB_MASK - 1)
```

The **MSB_MASK** and **SIZE_MASK** constant are used to extract the most significant bit from the other bits in the header. You're concerned that compilation with simply replicate these complex expressions every time they're used and that your throughput scores will suffer.

a) [2 points] Fortunately, **gcc** is smarter than that, even when compiled with flag **-O0**! What specific compiler optimization can **gcc** apply to avoid repeatedly recomputing the mask? If you don't remember the name of it, then just describe it.

> More constant folding—even more obvious than that with the **strlen** from Problem 2. Everything in **(1 << ((sizeof(Header) << 3) - 1))** can be evaluated at compile time, so the compiler does precisely that.

The **is_free** function is given a pointer to a block's header and returns **true** if the block is free, and **false** otherwise.

```
bool is_free(Header *hdr) {
    return  /* expr */ ;
}
```

b) [4 points] As you can see, **is_free** is missing a return expression. From the choices below, circle **all expressions** that could be used to do the right thing.

```
((*hdr & MSB_MASK) == MSB_MASK)

(*hdr & MSB_MASK)

~(*hdr ^ MSB_MASK)

*hdr < 0

*(signed char *)hdr < 0

*(int *)hdr < 0

(int)*hdr < 0
```

*The one boxed in dashes would work on big endian machines, but not the myths. However, if you circled it, we counted it as correct provided you also circled the one above it, since they would then both be correct for the same reasons.*

c) [3 points] Implement **get_blocksize**. Given a pointer to a block's header, the function returns the total number of bytes in the block.

```
size_t get_blocksize(Header *hdr) {
   return 1L << (*hdr & SIZE_MASK);
}
```

d) [5 points] Next, implement **get_neighbor**. Given a pointer to a block header, **get_neighbor** returns a pointer to the header of the neighboring block to the right. If this block is last in the heap, the function returns **NULL**.

```
Header_t *get_neighbor(Header *hdr) {
  Header *next = hdr + get_blocksize(hdr);
  if (next >= heap_start + heap_size) return NULL;
  return next;
}
```

The first 8 bytes of the payload of a free block are used to store a pointer to the header of the next block on the free list. Helper functions **get_next** and **set_next** read and write those links.

```
Header *get_next(Header *hdr);
void set_next(Header *first, Header *second);
```

e) [3 points] Implement the **set_next** function to link from **first** to **second** using a single call to **memcpy**. Restated, the actual address stored in **second** should be written into the leading bytes of the payload immediately following the header addressed by **first**. The third argument must be framed in terms of the **sizeof** operator.

```
void set_next(Header *first, Header *second) {
   memcpy(first + 1, &second, sizeof(void *));
}
```

f) [3 points] Actually, you heed the lessons from **lab7** and recall that calling **memcpy** is expensive and that there's a much simpler way to achieve the same effect without calling it. Reimplement **set_next** to link from **first** to **second** using a single pointer assignment, one or more type typecasts, and one or more pointer dereferences.

```
void set_next(Header *first, Header *second) {
  *(Header **)(first + 1) = second;
}
```

g) [4 points] The function **`remove_from_freelist`** is supplied with a pointer to a header currently within the free list. The partial implementation below searches for the entry on the list, and you should add the necessary code to remove the entry from the list. (You should use **`set_next`** and/or **`get_next`** when implementing **`remove_from_freelist`** and can assume both functions work perfectly.)

```
void remove_from_freelist(Header *to_remove) {
  Header *prev = NULL;
  Header *cur = free_list;
  while (cur != to_remove) {
    prev = cur;
    cur = get_next(cur);
  }
  Header *next = get_next(cur);
  if (prev == NULL)
    free_list = next;
  else
    set_next(prev, next);
}
```

h) [2 points] With this allocator, two adjacent blocks can only be coalesced if they are both free and have the same total block size. Why must they be the same size?

Blocks need to be a perfect power of two in size, and if you combine two neighboring blocks of different sizes (e.g., 16, 64), you get a size (e.g., 80) that isn't a power of two.

i) [6 points] Implement the **`myfree`** function. It should mark the block as free and add it to front of the free list. If the neighboring block to the right is free and same size, it should be coalesced into this one and the neighbor's header is removed from the free list. Attempt to coalesce only a single neighbor, not a sequence of neighbors. You may make use of any of the previous helper functions and can assume they work correctly.

```
void myfree(void *payload) {
  Header *hdr = (Header *)ptr - 1;
  *hdr |= MSB_MASK;
  set_next(hdr, free_list);
  free_list = hdr;

  Header *next = get_neighbor(cur);
  if (next != NULL && *hdr == *next) { // compares size and freed
    remove_from_freelist(next);
    (*hdr)++; // promote power by 1
  }
}
```

This allocator maintains the free list in a singly linked list, and **`callgrind`** shows traversing the list to be a performance bottleneck for both **`mymalloc`** (which traverses to find an appropriately sized block) and **`myfree`** (which traverses within **`remove_from_freelist`** when coalescing).

You brainstorm and come up with two possible strategies to combat the bottleneck: 1) upgrading the singly linked list to a doubly linked list, and 2) splaying the one singly linked free lists into many, many singly linked lists, where all nodes in any one list are the same size, e.g., one list threads through all blocks of size 16, a second threads through all blocks of size 32, a third threads through all blocks of size 64, and so forth.

j) [8 points] Consider each of the two strategies and discuss their relative pros and cons. Your analysis should answer each of the following questions:

- What is the added code complexity?
- What is the expected impact on the throughput of **`mymalloc`**? of **`myfree`**?
- How would utilization change? Would it? Why?

Strategy 1: doubly link the free list

- code changes: We need to include backward link in addition to forward link when adding to and removing from the free list.
- throughput: **`mymalloc`** unchanged, **`myfree`** goes from O(n) to O(1)
- utilization: The minimum block size increases from 16 to 32, since we need at least 16 bytes of payload now, plus the one byte for the header, rounder up to a perfect power of two.

Strategy 2: maintain many free lists, binned by size

- code changes: separate by powers of two, using log size from header bits as index into array of linked lists. The code to add and remove from a free list is otherwise unchanged.
- throughput: **`mymalloc`** was O(n) but now O(1), **`myfree`** was O(n) now O(n/#bins ~ 32)
- utilization: **`mymalloc`** now effectively operates as best fit and may result in less fragmentation