CS107 December 11th, 2023
Autumn 2023

# 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 telephone Jerry at 415-205-2242.

Good luck!

SUNet ID (@stanford.edu): ______________________________

Last Name: ______________________________

First Name: ______________________________

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

[signed] ______________________________

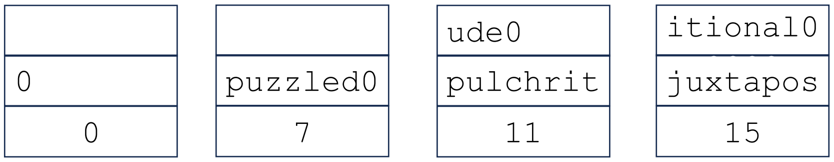
| | | Score | Grader |
|---|---|---|---|
| 1. Serializing Strings | [10] | ______ | ______ |
| 2. x86-64 and **`gcc`** optimizations | [20] | ______ | ______ |
| 3. Runtime Stack | [10] | ______ | ______ |
| 4. **`stop-and-copy`** Heap Compaction | [20] | ______ | ______ |
| **Total** | **[60]** | ______ | ______ |

**Problem 1: Serializing Strings [10 points]**

You're working with a custom string data type defined as follows:

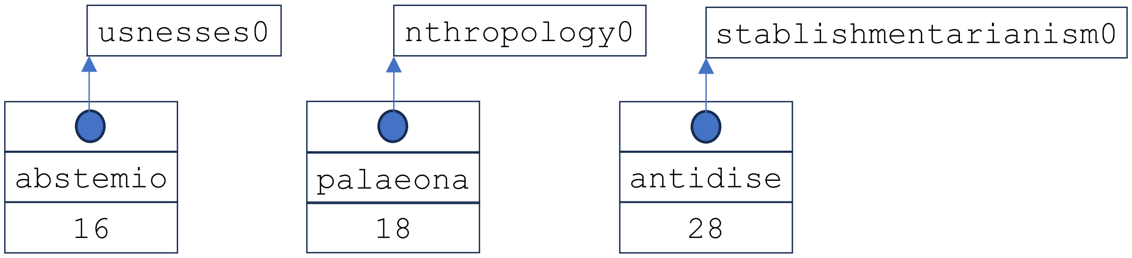
```
struct string {
   size_t length;
   char chars[16];
};
```

The string record is 24 bytes in size and allows for the storage of arbitrarily long strings, where the full length of the string is always stored in the **`length`** field. When the string itself is of length 15 or less, all characters—including the **`'\0'`** are stored in the **`string`**'s **`chars`** field. So, the strings **`""`**, **`"puzzled"`**, **`"pulchritude"`**, and **`"juxtapositional"`** could be represented in memory as:

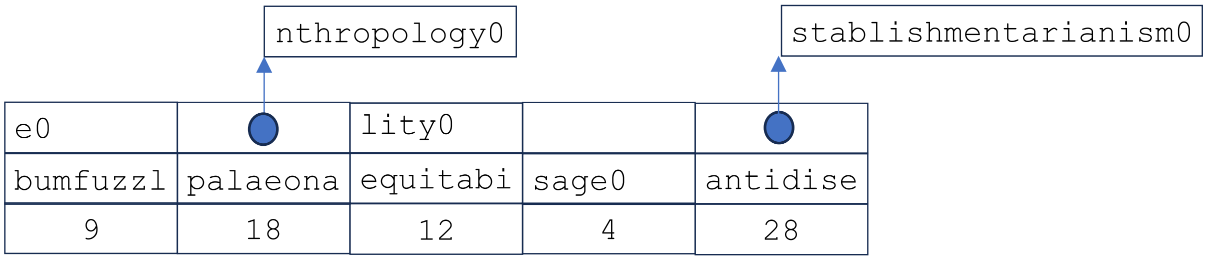


Note that I use **`0`** to represent a **`'\0'`**, and uninitialized characters or characters that don't matter are left as blank in the drawing.

When the string is of length 16 or more, the storage format is different. Specifically, the first 8 characters are stored in indices 0 through 7, inclusive, but the remaining eight characters—equivalently, the remaining **`sizeof(char *)`** bytes—collectively store the address of a traditional C string external to the struct. That C string stores all remaining characters—i.e., the characters at indices 8 and higher. So, **`"abstemiousnesses"`**, **`"palaeoanthropology"`**, and the infamous **`"antidisestablishmentarianism"`** would be represented as:



You're to implement a function called **`serialize`**, which accepts an array of these **`struct`**s and returns a single, **traditional**, dynamically allocated C string that's the ordered concatenation of all the strings stored in the array. So, given the following array of length 5:



your **`serialize`** function should return a dynamically allocated C string—namely, a heap-based C string:

```
"bumfuzzlepalaeoanthropologyequitabiltysageantidisestablishmentarianism"
```

Your implementation should make a single pass over the array, reallocating the initially **`strdup`**'ed empty string and appending the characters of each **`struct string`** in turn.

Use the rest of this page to provide your implementation:

```
char *serialize(struct string strings[], size_t length) {

   char *serialization = strdup("");
   // any other variable declarations below



   for (int i = 0; i < length; i++) {
      // complete the body of the for loop

   }

   return serialization;
}
```

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

The assembly code presented on the upper right was generated by compiling a function called **`allspice`** without optimization—i.e., using **`-Og`**.

```
0x1149 <+0>:   push   %rbp
0x114a <+1>:   push   %rbx
0x114b <+2>:   sub    $0x28,%rsp
0x114f <+6>:   mov    %rdi,%rbp
0x1152 <+9>:   mov    %rsi,0x8(%rsp)
0x1157 <+14>:  mov    (%rsi),%rbx
0x115a <+17>:  sub    %rdi,%rbx
0x115d <+20>:  mov    %rbx,0x18(%rsp)
0x1162 <+25>:  mov    0x18(%rsp),%rax
0x1167 <+30>:  add    %rax,%rax
0x116a <+33>:  cmp    %rbx,%rax
0x116d <+36>:  jbe    0x11af <allspice+102>
0x116f <+38>:  lea    0x18(%rsp),%rcx
0x1174 <+43>:  lea    0x8(%rsp),%rdx
0x1179 <+48>:  mov    %rbp,%rsi
0x117c <+51>:  mov    $0x1,%edi
0x1181 <+56>:  mov    $0x0,%eax
0x1186 <+61>:  callq  0x1050 <printf@plt>
0x118b <+66>:  mov    0x18(%rsp),%rax
0x1190 <+71>:  not    %rax
0x1193 <+74>:  test   $0x7,%al
0x1195 <+76>:  je     0x11af <allspice+102>
0x1197 <+78>:  mov    0x8(%rsp),%rsi
0x119c <+83>:  mov    0x8(%rsi),%rdi
0x11a0 <+87>:  callq  0x1149 <allspice>
0x11a5 <+92>:  mov    %rax,0x18(%rsp)
0x11aa <+97>:  add    %rax,%rbx
0x11ad <+100>: jmp    0x1162 <allspice+25>
0x11af <+102>: mov    0x18(%rsp),%rax
0x11b4 <+107>: add    $0x28,%rsp
0x11b8 <+111>: pop    %rbx
0x11b9 <+112>: pop    %rbp
0x11ba <+113>: retq
```

a) [12 points] First, fill in the blanks below so that **`allspice`** is programmatically consistent with the unoptimized assembly you see on the right. Note that the C code is nonsense and should just be a faithful reverse engineering of the assembly. You may not typecast anything.

*Note*: **`printf`** is a special function that expects its first three arguments to be passed through **`%rsi`**, **`%rdx`**, and **`%rcx`**, in that order. **`%rdi`** isn't used to pass traditional parameters when **`printf`** is called.

```
size_t allspice(char *mustard, char *cardamon[]) {

    size_t cinnamon = ________________________________;

    for (size_t i = ______________; _____________________; _______________) {

        printf(__________________, _______________, _______________);

        if (______________________) _______________;

        cinnamon = allspice(_______________, _______________);
    }

    return _____________;

}
```

Now, study the more aggressively optimized version of **`allspice`** presented on the right, and answer the questions below.

```
0x1170 <+0>:   push   %r13
0x1172 <+2>:   push   %r12
0x1174 <+4>:   push   %rbp
0x1175 <+5>:   push   %rbx
0x1176 <+6>:   sub    $0x28,%rsp
0x117a <+10>:  mov    (%rsi),%rbx
0x117d <+13>:  mov    %rsi,0x8(%rsp)
0x1182 <+18>:  sub    %rdi,%rbx
0x1185 <+21>:  lea    (%rbx,%rbx,1),%rax
0x1189 <+25>:  mov    %rbx,0x18(%rsp)
0x118e <+30>:  cmp    %rax,%rbx
0x1191 <+33>:  jae    0x11f0 <allspice+128>
0x1193 <+35>:  mov    %rdi,%rbp
0x1196 <+38>:  lea    0x18(%rsp),%r13
0x119b <+43>:  lea    0x8(%rsp),%r12
0x11a0 <+48>:  mov    %r12,%rdx
0x11a3 <+51>:  xor    %eax,%eax
0x11a5 <+53>:  mov    %r13,%rcx
0x11a8 <+56>:  mov    %rbp,%rsi
0x11ab <+59>:  mov    $0x1,%edi
0x11b0 <+64>:  callq  0x1050 <printf@plt>
0x11b5 <+69>:  mov    0x18(%rsp),%rax
0x11ba <+74>:  mov    %rax,%rdx
0x11bd <+77>:  not    %rdx
0x11c0 <+80>:  and    $0x7,%edx
0x11c3 <+83>:  je     0x11e4 <allspice+116>
0x11c5 <+85>:  mov    0x8(%rsp),%rsi
0x11ca <+90>:  mov    0x8(%rsi),%rdi
0x11ce <+94>:  callq  0x1170 <allspice>
0x11d3 <+99>:  add    %rax,%rbx
0x11d6 <+102>: lea    (%rax,%rax,1),%rdx
0x11da <+106>: mov    %rax,0x18(%rsp)
0x11df <+111>: cmp    %rbx,%rdx
0x11e2 <+114>: ja     0x11a0 <allspice+48>
0x11e4 <+116>: add    $0x28,%rsp
0x11e8 <+120>: pop    %rbx
0x11e9 <+121>: pop    %rbp
0x11ea <+122>: pop    %r12
0x11ec <+124>: pop    %r13
0x11ee <+126>: retq
0x11ef <+127>: nop
0x11f0 <+128>: add    $0x28,%rsp
0x11f4 <+132>: mov    %rbx,%rax
0x11f7 <+135>: pop    %rbx
0x11f8 <+136>: pop    %rbp
0x11f9 <+137>: pop    %r12
0x11fb <+139>: pop    %r13
0x11fd <+141>: retq
```

b) [2 points] Note that both the unoptimized and optimized versions some caller-owned registers to the stack but then proceeds to modify **`%rsp`** without having pushed its value to the stack. However, you also know that **`%rsp`** is caller-owned as well, even though it's not being pushed to the stack. Why does, say, **`%rbp`** needs to be pushed to the stack while **`%rsp`** doesn't need to be?

c) [2 points] It should be evident that the number of instructions emitted when **`-O2`** is used is much higher than the number emitted with **`-Og`**. Explain why the number of instructions executed by a typical call to **`allspice`** will, in practice, still be smaller.

d) [2 points] Notice that 5 of the final 6 instructions appear earlier, in the same order, at offsets **`+116`** through **`+126`**, and that the **`mov %rbx, %rax`** at offset **`+132`** is the only one not replicated. Knowing that you could, in theory, implement the **`allspice`** directly in x86-64 yourself, by hand, explain how you might use most of what's presented to the right (which is a copy of what's on the previous page) while reordering a few instructions and updating one or more jump offsets so that only one copy of these five instructions is needed instead of two.

```
0x1170 <+0>:   push   %r13
0x1172 <+2>:   push   %r12
0x1174 <+4>:   push   %rbp
0x1175 <+5>:   push   %rbx
0x1176 <+6>:   sub    $0x28,%rsp
0x117a <+10>:  mov    (%rsi),%rbx
0x117d <+13>:  mov    %rsi,0x8(%rsp)
0x1182 <+18>:  sub    %rdi,%rbx
0x1185 <+21>:  lea    (%rbx,%rbx,1),%rax
0x1189 <+25>:  mov    %rbx,0x18(%rsp)
0x118e <+30>:  cmp    %rax,%rbx
0x1191 <+33>:  jae    0x11f0 <allspice+128>
0x1193 <+35>:  mov    %rdi,%rbp
0x1196 <+38>:  lea    0x18(%rsp),%r13
0x119b <+43>:  lea    0x8(%rsp),%r12
0x11a0 <+48>:  mov    %r12,%rdx
0x11a3 <+51>:  xor    %eax,%eax
0x11a5 <+53>:  mov    %r13,%rcx
0x11a8 <+56>:  mov    %rbp,%rsi
0x11ab <+59>:  mov    $0x1,%edi
0x11b0 <+64>:  callq  0x1050 <printf@plt>
0x11b5 <+69>:  mov    0x18(%rsp),%rax
0x11ba <+74>:  mov    %rax,%rdx
0x11bd <+77>:  not    %rdx
0x11c0 <+80>:  and    $0x7,%edx
0x11c3 <+83>:  je     0x11e4 <allspice+116>
0x11c5 <+85>:  mov    0x8(%rsp),%rsi
0x11ca <+90>:  mov    0x8(%rsi),%rdi
0x11ce <+94>:  callq  0x1170 <allspice>
0x11d3 <+99>:  add    %rax,%rbx
0x11d6 <+102>: lea    (%rax,%rax,1),%rdx
0x11da <+106>: mov    %rax,0x18(%rsp)
0x11df <+111>: cmp    %rbx,%rdx
0x11e2 <+114>: ja     0x11a0 <allspice+48>
0x11e4 <+116>: add    $0x28,%rsp
0x11e8 <+120>: pop    %rbx
0x11e9 <+121>: pop    %rbp
0x11ea <+122>: pop    %r12
0x11ec <+124>: pop    %r13
0x11ee <+126>: retq
0x11ef <+127>: nop
0x11f0 <+128>: add    $0x28,%rsp
0x11f4 <+132>: mov    %rbx,%rax
0x11f7 <+135>: pop    %rbx
0x11f8 <+136>: pop    %rbp
0x11f9 <+137>: pop    %r12
0x11fb <+139>: pop    %r13
0x11fd <+141>: retq
```

e) [2 points] At offset +51, you'll see an **`xor`** instruction. What line in the unoptimized version does that correspond to? In what sense is the **`xor`** alternative considered an optimization?

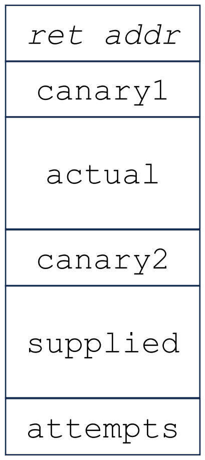
**Problem 3: Runtime Stack [10 points]**

**a)** [5 points] You've been hired as a security engineer to scrutinize some crucial authentication code to see how someone might access a website even though they don't have the proper credentials to do so. The key function to examine is below:

```
#define MAX_ATTEMPTS 3
bool login() {
    size_t canary1 = rand_size_t(); // generates random size_t
    char actual[16];
    size_t canary2 = canary1;
    char supplied[16];

    strcpy(actual, retrieve_password()); // assume no issues
    size_t attempts = 0;
    while (attempts < MAX_ATTEMPTS) {
        printf("Enter password: ");
        gets(supplied);
        if (canary1 != canary2) {
            printf("Hacking attempt! Aborting login!\n");
            return false;
        }
        if (strncmp(supplied, actual, 16) == 0) return true;
        printf("Supplied password failed. Try again!\n");
        attempts++;
    }
    printf("Max attempts exceeded.\n");
    return false;
}
```

Note the truly daft use of **`gets`**, which reads a line from **`stdin`** and places whatever is typed into the **`supplied`** character array, where no checks for overflow are performed whatsoever. In the context of **`login`**, if the user types in the seven-letter **`"abcdefg"`** or even the 15-letter **`"abcdefghijklmno"`**, there's enough space for the 8 or 16 bytes needed to store them. However, if the user types in the full lowercase alphabet, 27 bytes would be written to flood the **`supplied`** array and overflow beyond its last allocated byte.



You admire, however, the use of ***canaries***, which are special values used to detect buffer overflow. In this case, **`canary1`** and **`canary2`** are set to the same randomly generated 64-bit value. The assumption here is that any overflow of **`supplied`**—intentional or not—changes **`canary2`** and prompts **`login`** to return **`false`** right away after printing a panicky message.

By examining the assembly code for the **`login`** function, you determine the five local variables are laid out as presented as drawn in the diagram on the right. The smaller rectangles are 8-byte **`size_t`**s, larger rectangles are 16-byte

**`char`** arrays. And note that the return address—that is, the saved rip value necessary for function call and return to work—rests right on top of the stack frame.

Referring to the **`login`** implementation and the stack diagram of locals, you explain to those who've hired you that someone can be granted access to the website—that is, fool **`login`** to return **`true`**—regardless of whatever **`retrieve_password`** returns (save that it returns a C string of length 15 or less, which you can assume to be the case).

- [3 points] You have the idea of entering 47 **`'0'`**s as the first password and then 23 **`'0'`**s as the second password, and you try. How far do you get? Leveraging your understanding of how the stack organizes **`login`**'s local variables as per the diagram on the prior page, explain why this won't work.

- [2 points] Now explain why a single supplied password of 48 **`'a'`**'s in a row might allow **`login`** to stage a **`true`** as a return value, only to crash or cause other problems as it attempts to pass control back to the caller. But also explain why entering 48 **`'a'`**'s might actually work without a problem, and what must be true for that to happen.

**b)** [5 points] You learn of a nifty **`get_function_by_address`** with the following prototype:

```
struct func_info {
    char *name;     // e.g., main, printf, mymalloc, level_1, etc.
    void *start;    // address of first x86-64 instruction
    void *end;      // address of last x86-64 instruction
};

bool get_function_by_address(void *addr, struct func_info *info);
```

**`get_function_by_address`** accepts an arbitrary memory address and returns **`true`** if and only if the supplied address is that of an assembly code instruction in the code segment, and **`false`** otherwise. When **`true`** is returned, the record addressed by **`info`** is populated with information about the function itself. (When **`false`** is returned, the **`struct`** addressed by **`info`** is unchanged.)

You're inspired to implement a **`backtrace`** function to imitate the functionality of **`gdb`**'s **`backtrace`** command. Your **`backtrace`** function identifies the sequence of currently active function calls that have yet to return and prints their names. This is your implementation:

```
void backtrace() {
    struct func_info info;
    void **start = &info;
    while (true) {
        if (get_function_by_address(*start, &info)) {
            printf("%s\n", info.name);
            if (strcmp(info.name, "main") == 0) return;
        }
        start++;
    }
}
```

So, if **`main`** calls **`foo`** calls **`bar`** calls **`baz`** calls **`pop`** calls **`backtrace`**, the call to **`backtrace`** would print:

```
pop
baz
bar
foo
main
```

and then return.

After studying the implementation on the prior page, answer the following questions, limiting your responses to at most 75 words.

- [3 points] Briefly explain how and why the **`backtrace`** function works. Specifically, explain what types of values on the stack prompt the **`get_function_by_address`** to return **`true`**, and also explain why **`start`** is initialized to **`&info`**.

- [2 points] You're convinced the general implementation is pretty good, but then you realize that **function pointers** can be passed as parameters and stored in local variables, and you don't want the names of the functions they address to be printed just because they're stored on the stack. Describe how you might change the implementation of **`get_function_by_address`** to only print the names of functions that are currently executing without printing the names of functions simply function pointers are passed as parameters or stored as local variables.

**Problem 4: `stop-and-copy` Heap Compaction [20 points]**

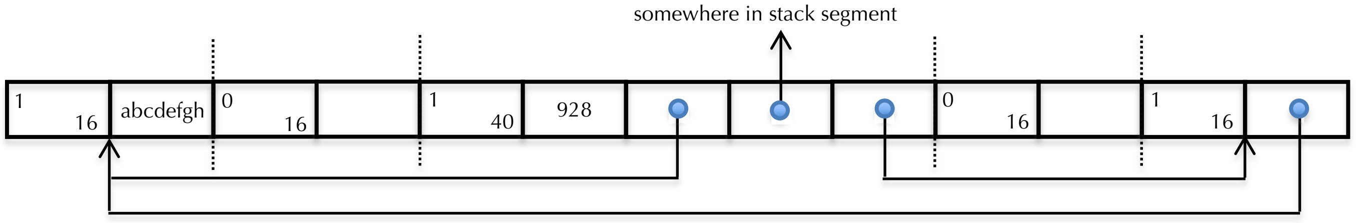
Some custom allocators optimize to solve the heap compaction problem by maintaining **two** heaps instead of one. Only one heap is active at any one moment, but when heap compaction is invoked, the ordered concatenation of all allocated blocks is written to the inactive heap, all internal pointers aliasing active heap memory are rewritten to reference the inactive heap clone, and then the active heap is rendered inactive and vice versa.

All pointers—whether stored in the heap, on the stack, or in global variables—would need to be updated, but here we'll assume all pointers into the active heap are stored within the active heap itself and nowhere else and that all such pointers always address the first byte of an allocated node's payload. Throughout the problem, we'll assume we're working with a 64-bit system where all **`size_t`**'s and pointers are eight bytes long. The active heap is **`sizeof(size_t)`**-aligned, is subdivided into nodes that are always a multiple of eight bytes, and nodes always have enough space for at least eight bytes of payload. The header is an eight-byte **`size_t`**, where the most significant bit is 1 for allocated nodes and 0 for free nodes, and the other 63 bits encode the node size, in bytes—that is, the header size plus the payload size, which is understood to be a multiple of 8.

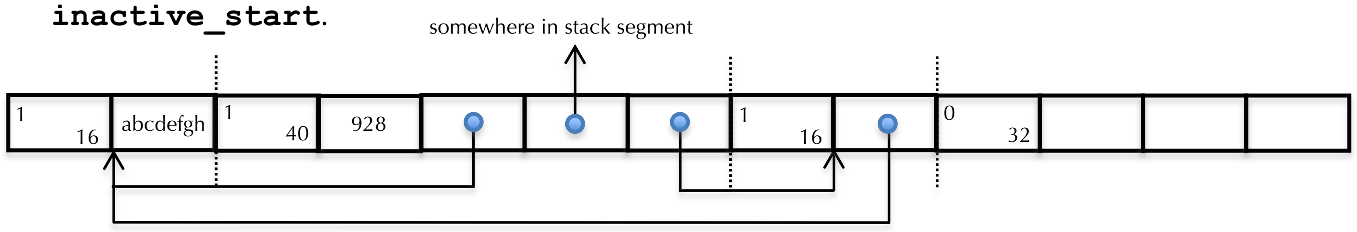
The free list is always implicit, so there are no next or previous pointers to worry about. For simplicity, we'll assume each of the two heaps are the same size, and that the bases of the two heaps are stored in global variables, as with:

```
static size_t *active_start;
static size_t *inactive_start;
#define HEADER_SIZE sizeof(size_t);
#define HEAP_SIZE (1L << 28) // 256 megabytes
```

The goal of this problem is to take a picture like so, the base address of which is stored **`active_start`**:



and replicate the in-use nodes to look like this, the base address of which is stored in **`inactive_start`**.



The whole process looks like a loop of **`memcpy`** calls, but with the added complexity that comes because pointers in the payloads of active nodes that appear to be active heap addresses need to be recalculated when dropped in the inactive-but-soon-to-be-made-active heap.

For this problem, you'll implement this heap compaction algorithm, affectionately known as the **stop-and-copy** algorithm. You'll be led through a series of steps—less efficient than it could be, but easier to do if carefully managed across several steps.

**a)** [3 points] Implement the **`node_is_allocated`** predicate function, which accepts the address of a node header as a **`size_t *`** and returns **`true`** if and only if the node is allocated. Recall that the most significant bit is 1 for allocated nodes and 0 for free nodes. Your implementation should be very short.

```
bool node_is_allocated(size_t *header) {
```

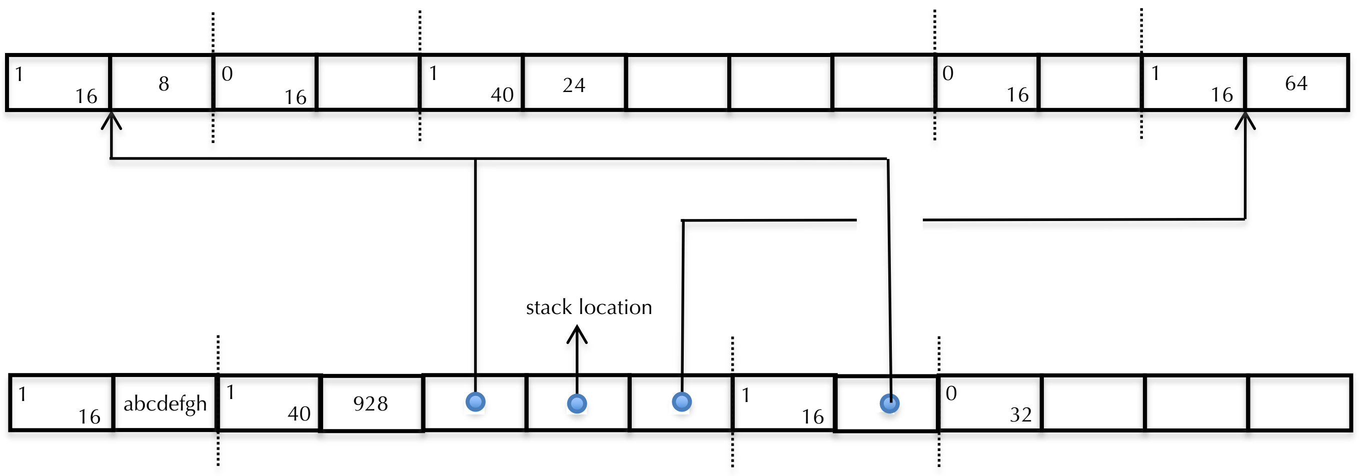
**b)** [3 points] Implement the **`node_get_size`**, which accepts the address of a node header as a **`size_t *`** and returns the number of bytes—header and payload included—making up the node. Note that the node may or may not be allocated. This should also be very short.

```
size_t node_get_size(size_t *header) {
```

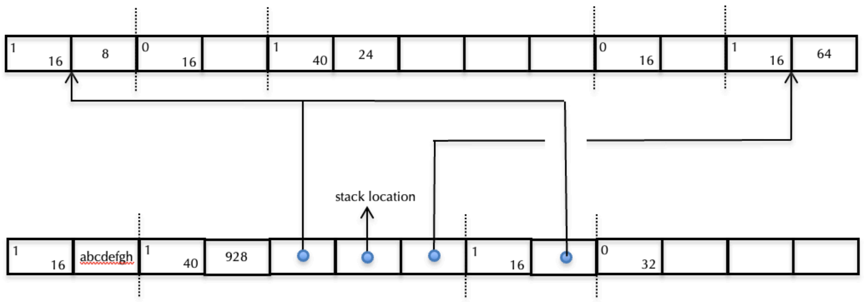
**c)** [8 points] Implement the **`replicate_active_heap`** function, which verbatim replicates all of the active heap's allocated nodes—and just the allocated nodes—by copying them, in order, to the inactive heap, and then taking care to properly mark the last node of the inactive heap as unallocated.

While you shouldn't worry about rewriting the pointers within the payloads just yet, you **should** update the first eight bytes of each active heap node's payload to store the offset, in bytes, from the base address of the inactive heap where the corresponding payload was copied. In the figure below, note that each of the eight-byte words following allocated node headers in the active heap have been updated with the numbers 8, 24, and 64. That's because their payloads have been copied to addresses **`inactive_heap + 8`**, **`inactive_heap + 24`**, and **`inactive_heap + 64`**, respectively.

Place your implementation on the next page. The above problem statement has been repeated on the next page as well in a smaller font to you needn't flip back and forth between that page and this one.



*The above figure reflects the state of the active heap (top) and the inactive heap (bottom) after the* **`replicate_active_heap`** *function you're implementing for part c has executed. The empty boxes in the active heap aren't filled in because their contents don't matter after everything has been replicated.*



Implement the **replicate_active_heap** function, which verbatim replicates all of the active heap's allocated nodes—and just the allocated nodes—by copying them, in order, to the inactive heap, and then taking care to properly mark the last node of the inactive heap as unallocated.

While you shouldn't worry about rewriting the pointers within the payloads just yet, you **should** update the first eight bytes of each active heap node's payload to store the offset, in bytes, from the base address of the inactive heap where the corresponding payload was copied. In the figure on the right, note that each of the eight-byte words following allocated node headers in the active heap have been updated with the numbers 8, 24, and 64. That's because the payloads have been copied to addresses **inactive_heap + 8**, **inactive_heap + 24**, and **inactive_heap + 64**, respectively.

```
void replicate_active_heap() {
```

**d)** [6 points] Finally, implement the **`rewrite_addresses`** function, which walks through all the payload words in the inactive heap, and rewrites any pointer addressing a word in the active heap with the corresponding address in the inactive heap. If the eight-byte figure doesn't look like a pointer within the active heap—that is, it's outside the regional between **`active_start`** and some **`active_end`** you'll need to compute—then you can ignore it and simply move onto the next eight bytes of payload. Note that if a figure looks to be an address within the active heap, then it is guaranteed to be the base address of an allocated node's payload.

```
void rewrite_addresses() {
```