CS107 December 9th, 2024

Autumn 2024

CS107 Final Examination Solution

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: _____

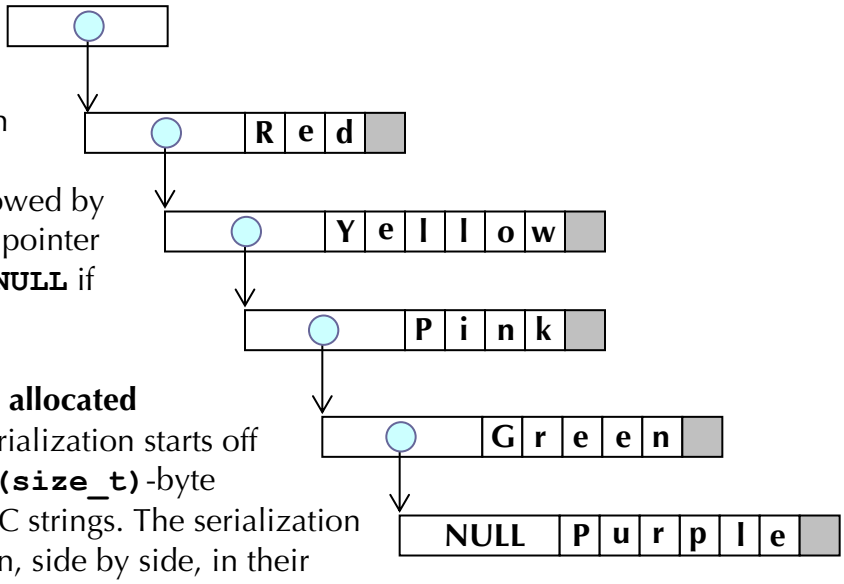
I accept the letter and spirit of Stanford's Honor Code.

[signed] _____

		Score	Grader
1. C Strings and Memory Manipulation	[10]	_____	_____
2. Generics and Function Pointers	[10]	_____	_____
3. x86-64 and **`gcc`** optimizations	[20]	_____	_____
4. Runtime Stack	[10]	_____	_____
5. Heaps with Headers and Footers	[30]	_____	_____
Total	**[80]**	_____	_____

Solution 1: C Strings and Memory Manipulation [10 points]

Write a function **`serialize`** to convert a linked list to a single stream of **`'\0'`**-delimited characters arrays.



The list consists of a series of variably sized nodes, where each node stores an eight-byte pointer, followed by the individual characters of a C string, followed by a **`'\0'`** (drawn as shaded boxes). Each pointer stores the address of the next node, or **`NULL`** if there aren't any more.

`serialize` synthesizes a **dynamically allocated** serialization of the supplied list. The serialization starts off with a dynamically allocated, **`sizeof(size_t)`**-byte figure ultimately storing the number of C strings. The serialization then continues to lay the C strings down, side by side, in their original order. If handed the above list, **`serialize`** would build and return the base address of the **`size_t`** storing **`5`**:

`serialize` takes the address of the first node (the one storing **`"Red"`** in the diagram above, for example) as a **`void *`** and constructs the serialization. Your implementation should be implemented in a **single pass** over the list, **`realloc`**ating as necessary, without freeing the nodes of the original list. You should return the base address of the entire figure as a **`size_t *`**, and your function should work for any properly structured list, including the empty list.

Here's my own implementation!

```
size_t *serialize(void *list) {
   size_t *serialization = malloc(sizeof(size_t));
   size_t length = sizeof(size_t);
   void **curr = (void **) list;
   size_t count = 0;

   while (curr != NULL) {
      char *str = (char *)(curr + 1);
      serialization = realloc(serialization, length + strlen(str) + 1);
      strcpy((char *) serialization + length, str);
      length += strlen(str) + 1;
      curr = (void **) *curr;
      count++;
   }

   *serialization = count;
   return serialization;
}
```

Solution 2: Generics and Function Pointers [10 points]

Write a generic function **`eliminate_largest_gap`** that accepts any sorted array of elements and removes the two neighboring elements that are furthest from one another compared to all other neighboring pairs, sliding higher-index elements that remain in to close the gap. For instance, given the following **`int`** array:

-2	3	5	8	11	17	20	24

the function would spot the 11 and 17 as furthest apart and remove both, leaving the array as:

-2	3	5	8	20	24

If there's a tie for largest gap between two or more pairs, you should remove the pair residing at the lowest pair of neighboring indices. Of course, we want our function to be generic, and that's possible provided we're passed the array element size and the appropriate comparison function.

a) [6 points] In the space below, present the implementation of your **`eliminate_largest_gap`** function. You can assume the array is sorted from low to high according to the comparison function. Recall that the return value of the comparison function is a statement of just how different the two items being compared are.

```
void eliminate_largest_gap(void *base, size_t length, size_t elem_size,
                           int (*cmpfn)(const void *, const void *)) {
    assert(length >= 2);
    size_t offset = 0;
    int largest_diff = 1;
    for (size_t i = 0; i < length - 1; i++) {
        void *left = (char *) base + i * elem_size;
        void *right = (char *) base + (i + 1) * elem_size;
        int diff = cmpfn(left, right);
        if (diff < largest_diff) {
            offset = i;
            largest_diff = diff;
        }
    }

    void *start = (char *) base + offset * elem_size;
    void *stop = (char *) start + 2 * elem_size;
    size_t bytes = (length - offset - 2) * elem_size;
    memmove(start, stop, bytes);
}
```

b) [2 points] Generic functions like that from part a), when operating on arrays of **`char *`**s, require a comparison function like that presented below if the strings are to be compared alphabetically.

```
int string_compare(const void *one, const void *two) {
    return strcmp(*(char **)one, *(char **)two);
}
```

Briefly explain why **`one`** and **`two`** need to be cast to **`char **`**'s here instead of just **`char *`**'s.

The items passed to generic comparison functions are always addresses of array elements, because that's the only thing the generic is capable of computing for each of the elements. Because the elements are **`char *`**'s, their addresses are truly **`char **`**'s. We use the cast to tell the truth about what the incoming elements really are. Since they're truly **`char **`**'s, that's what the cast should be.

c) [2 points] You have a sorted array of **`struct student`**s called **`students`**, where:

```
struct student {
    char *name;       // standard C string
    char sunet[8];    // SUNet @stanford email address, limited to 8 character
    // other fields
};
```

defines the type of interest. What happens if you call

```
eliminate_largest_gap(students, n, sizeof(struct student), string_compare);
```

Note that **`students`** is a length-**`n`** array of records, but that we're passing in **`string_compare`** for the comparison function. Assuming the array is well-formed, what will happen when this is executed? Will it crash? If so, why? Or will it execute without crashing? If so, what will it do, and why?

This will work and remove the two structs with name fields that are furthest away from each other according to **`string_compare`**. The base addresses of **`struct student`**s are passed to **`string_compare`**, but the base address of the **`struct`** is also the base address of its first field, which is the **`char *`** called **`name`**. So the base address is as much a **`char **`** as it is a **`struct student *`**.

Solution 3: x86-64 and gcc Optimizations [20 points]

The assembly code presented on the lower right was generated by compiling the **cheese** function without optimization using **-Og**. (Note: the subparts only added up to 18 points, so we invented a part e while grading and gave everyone two out of two points on it.)

a) [12 points] First, fill in the blanks below so that **cheese** 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 compilation. You may not typecast anything.

```
1129: sub    $0x38,%rsp
112d: mov    %rdi,0x18(%rsp)
1132: mov    %rsi,0x10(%rsp)
1137: mov    %rdx,0x8(%rsp)
113c: mov    0x18(%rsp),%rax
1141: lea    0x0(, %rax,8),%rdx
1149: mov    0x8(%rsp),%rax
114e: add    %rdx,%rax
1151: mov    (%rax),%rdx
1154: mov    0x10(%rsp),%rax
1159: sub    %rax,%rdx
115c: mov    %rdx,%rax
115f: mov    %rax,0x28(%rsp)
1164: mov    0x28(%rsp),%rax
1169: and    $0x3,%eax
116c: cmp    $0x2,%rax
1170: je     11ad <cheese+0x84>
1172: mov    0x8(%rsp),%rax
1177: add    $0x8,%rax
117b: mov    (%rax),%rcx
117e: lea    0x10(%rsp),%rdx
1183: mov    0x18(%rsp),%rax
1188: mov    %rcx,%rsi
118b: mov    %rax,%rdi
118e: callq  1129 <cheese>
1193: mov    0x10(%rsp),%rax
1198: test   %rax,%rax
119b: je     11b0 <cheese+0x87>
119d: mov    0x28(%rsp),%rax
11a2: shr    $0x3,%rax
11a6: mov    %rax,0x28(%rsp)
11ab: jmp    1172 <cheese+0x49>
11ad: nop
11ae: jmp    11b1 <cheese+0x88>
11b0: nop
11b1: add    $0x38,%rsp
11b5: retq
```

```
void cheese(size_t swiss, char *gouda, char **cheddar) {

    size_t blue = cheddar[swiss] - gouda;

    if ((blue & 0x3) == 0x2) return;

    while (true) {

        cheese(swiss, cheddar[1], &gouda);

        if (gouda == NULL) break;

        blue /= 8;
    }
}
```

Now, study the aggressively optimized version of **cheese** on the right and answer the questions below.

```
11a0: push  %r13
11a2: push  %r12
11a4: push  %rbp
11a5: push  %rbx
11a6: sub   $0x18,%rsp
11aa: mov   (%rdx,%rdi,8),%rax
11ae: mov   %rsi,0x8(%rsp)
11b3: sub   %rsi,%rax
11b6: and   $0x3,%eax
11b9: cmp   $0x2,%rax
11bd: je    11e1 <cheese+0x41>
11bf: mov   0x8(%rdx),%r13
11c3: mov   %rdi,%rbp
11c6: mov   %rsi,%rbx
11c9: lea   0x8(%rsp),%r12
11ce: mov   %r12,%rdx
11d1: mov   %r13,%rsi
11d4: mov   %rbp,%rdi
11d7: callq 11a0 <cheese>
11dc: test  %rbx,%rbx
11df: jne   11ce <cheese+0x2e>
11e1: add   $0x18,%rsp
11e5: pop   %rbx
11e6: pop   %rbp
11e7: pop   %r12
11e9: pop   %r13
11eb: retq
```

b) [2 points] Note that the optimized version allocates a smaller number of bytes on the stack than the unoptimized version does. What is the optimized version doing that allows it to operate using less stack space?

The optimized version saves the contents of four callee-saved registers, **r13**, **r12**, **rbp**, and **rbx**. That frees up these four registers to be used to store temporary values in registers instead of the stack frame.

c) [2 points] Recall that strength reduction is an optimization technique where expensive arithmetic instructions are replaced with computationally equivalent (but less expensive) instructions. Interestingly enough, the **unoptimized** version shows clear evidence that strength reduction was used in in at least one place. Identify which of all the instructions is the result of a strength reduction?

The most obvious strength reduction? Going with **shr** by **3** instead of **div** by **8**. You could argue that lea instructions are being used to do math in a single assembly code instruction, avoiding the cascade of many exposed **add** and **mul** instructions. A less obvious one is the **test %rax,%rax** instructions used instread of the more obvious **cmp $0,%rax**. Both are single instructions, but the former requires a three-byte encoding instead of the latter's five-byte encoding. Note we only expected one example of a strength reduction, and the **shr**-instead-of-**div** is the one we expected to see.

d) [2 points] Note that the **shr** instruction you see in the unoptimized version is altogether missing in the optimized version. Why is it gone?

Because **blue** doesn't influence control flow after the first **if** test, the compiler sees there's no reason to emit code for the last line that updates **blue**. It recognizes that **blue**'s value doesn't matter and pretends as if **blue** doesn't even exist within the loop. This is a form of **dead code elimination**, though you didn't need to know the phrase.

Solution 4: Runtime Stack [10 points]

You're working as a security engineer for a company that maintains an open-source version of a terminal-based chat client akin to WhatsApp or iPhone Messages, and you're seeing reports from ethical hackers that one can gain access to any account whatsoever. You know there's a backdoor password—a very long, obscure one—that works for any account, but you're told it's possible to gain access to any account using any one of a boundless number of passwords without any knowledge of the backdoor password. You're assigned the task of figuring out what's up. You have the source and have isolated the issue down to **`authenticate`**, presented right here:

```
bool authenticate(char *supplied_password) {
    char secret[16];
    load_secret_password(secret, sizeof(secret));
    char supplied[16];
    strcpy(supplied, supplied_password);
    return strncmp(secret, supplied, sizeof(supplied)) == 0;
}
```

You know not to question the **`load_secret_password`** function, because that's scrutinized daily to ensure it's 100% secure. So, you fire up **`gdb`** to investigate what happens when you call **`authenticate("doris")`**. Here's the compact summary of what you find to be important.

```
Breakpoint 1, authenticate (supplied_password=0x7fffffffeb02 "doris")
13    bool authenticate(const char *supplied_password) {
(gdb) p/x $rsp
$1 = 0x7fffffffe718
(gdb) disassemble authenticate
Dump of assembler code for function authenticate:
   0x00005555555551df <+0>:   sub    $0x38,%rsp
   .. other instructions not important
(gdb) p/x $rip
$2 = 0x5555555551df
(gdb) ni
(gdb) p/x $rsp
$3 = 0x7fffffffe6e0
(gdb) p/x &secret[0]
$4 = 0x7fffffffe700
(gdb) p/x &supplied[0]
$5 = 0x7fffffffe6f0
```

In particular, you notice the compiler gives you 56 bytes of stack space even though you only need 32 bytes for the two local arrays! There are 8 bytes of padding *above* **`secret`** and 16 bytes of padding *below* **`supplied`**. The compiler often over-allocates stack space so that the stack pointer—that is, the address value tracked by **`%rsp`**—is a multiple of 32 bytes.

a) [4 points] By examining the C code and the **gdb** output, give an example of a password that's sure to fool **authenticate** into returning **true**, and explain why your example works.

It's clear from the **gdb** snooping that **secret** layers right on top of **supplied**, with no padding in between. That means a call to **strcpy**, where **supplied** is the first argument, will copy characters into **secret** if the source string is long enough. And if you choose a string of length 32, where the first 16 characters are repeated verbatim as the last 16 characters, **supplied** and **secret** will match as far as that **strncmp** call is concerned. (Of course, a string of length 32 is a stream of 33 characters, because of that **'\0'** at the end. That zero byte would be replicated in the first byte of padding above **secret**, but because it's padding, it won't damage the program.)

b) [3 points] You also notice when you type in a password of length 128, the chat client crashes as it returns from **authenticate**. Very clearly explain why such a long password interferes with **authenticate**'s ability to return back to whatever function called it.

The stack frame for **authenticate**'s local variables is much smaller than 128! When a string of length 128 is **strcpy**'ed into **supplied**, it floods **supplied**, **secret**, the padding above **secret**, and the memory above that padding with characters. Unfortunately, the eight bytes directly above the padding hold the return **%rip** address. By overwriting those eight bytes with random content, you overwrite a legitimate address instruction with a bogus one with a farcically high probability, thereby interfering with the program's ability to jump back to the call site.

c) [3 points] Provide a simple, one or two-line fix for **authenticate** that will prevent both the unauthorized access security and the crash.

The simplest thing to do is to truncate the **supplied_password** argument if it's of length 16 or more before any **string.h** functions are called, as with:

```
if (strlen(supplied_password) > 15)
   supplied_password[16] = '\0';
```

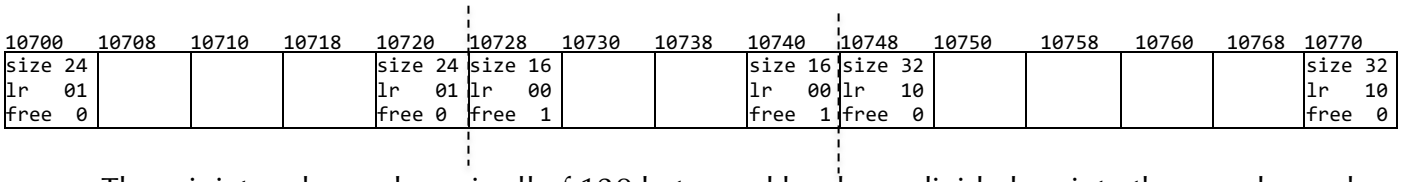
There are certainly other solutions as well, but the one above is the most straightforward.

Solution 5: Heaps with Headers and Footers [30 points]

You are implementing a custom allocator that relies on an eight-byte header and an eight-byte footer—a replica of the same node's header—that occupies the last eight bytes of a node, whether that node is free or not.

The size of a node's payload is always a multiple of 8 and at least 16 bytes, and the payload size—that it, the size of the node minus the 16 bytes allotted to the header and footer—is stored in the header as a **`size_t`**. Because all payload sizes are constrained to be multiples of 8, we can use the three least significant bits to store allocation status of the node and its left and right neighbors. Specifically:

- The **least** significant bit of a header/ footer records whether the node is free (0) or in use (1)
- The **second** least significant bit records whether the node to the **right** is free (0) or in use (1)
- The **third** least significant bit records whether the node to the **left** is free (0) or in use (1).



The miniature heap above is all of 120 bytes and has been divided up into three nodes, only one of which—the one residing at address 0x10728, is in use. That implies the client is storing that middle node's payload address of 0x10730 in a variable somewhere and using its 16 bytes to store whatever it wants to there.

Assume the following types, constants, and globals have been defined:

```
typedef size_t Header;
typedef Header Footer;

#define HEADER_SIZE sizeof(Header)
#define FOOTER_SIZE sizeof(Footer)
#define MIN_PAYLOAD_SIZE (2 * sizeof(size_t))
#define MIN_NODE_SIZE (HEADER_SIZE + MIN_PAYLOAD_SIZE + FOOTER_SIZE)

static void *segment_start;
static void *segment_end;
static size_t segment_size;

#define LEFT_FREE (1UL << 2)
#define RIGHT_FREE (1UL << 1)
#define IN_USE_MASK (1UL << 0)
#define SIZE_MASK (~(IN_USE_MASK | LEFT_FREE | RIGHT_FREE))

#define FREE 0
#define ALLOCATED 1
```

a) [6 points, 2 points per line] You'll first want to write two utility functions—**set_payload_size** and **set_free_status**—that know how to update the payload size and allocation status within a node's header (but not the footer) without overwriting any other information. In particular, **set_payload_size** overwrites the upper 61 bits with the upper 61 bits of the supplied size, **set_free_status** sets the least significant bit accordingly while leaving the other 63 bits as is. Be sure that neither changes the two bits dedicated to the neighbors' allocation status. Write your one-liners directly over the blanks.

```
void set_payload_size(Header *header, size_t size) {
    assert(size % sizeof(size_t) == 0);
    assert(size >= MIN_PAYLOAD_SIZE);
    // one line
    *header = size | (*header & ~SIZE_MASK);
}

void set_free_status(Header *header, size_t status) {
    assert(status <= ALLOCATED);
    // just one single-line expression
    if (status == ALLOCATED) *header |= IN_USE_MASK;
    // one more single-line expression
    else if (status == FREE) *header &= ~IN_USE_MASK;
}
```

b) [3 points] Now that you have to two functions above, you're in a position to implement a function—uncreatively called **set_size_and_free_status**—that updates both the header and the footer of the node beginning at the supplied address, without disturbing the bits used to track left and right neighbor allocation status. The code supplied overwrites an existing header with a new size and allocation status, and you're to fill in the remaining two lines that, based on the new size information, locates the footer so its contents can be set to be a clone of the new header.

Fill in the one line below to compute the address of the footer, **subject to the constraint that you mustn't use any explicit typecasts anywhere in your expression**. (Note that **Header**, **Footer**, and **size_t** are all really the same type and can be used interchangeably.)

```
void set_size_and_free_status(Header *header, size_t payload, size_t status) {
    set_payload_size(header, payload);
    set_free_status(header, status);

    Header *footer = header + 1 + payload/HEADER_SIZE;
    memcpy(footer, header, sizeof(Header));
}
```

c) [2 points] The last line of **set_size_and_free_status** correctly uses **memcpy** in the sense that it accurately replicates the header bit pattern into the space addressed by **footer**. Calling **memcpy**, however, is a foolishly expensive way to replicate such a small number of bits and would dramatically impact the throughput of your allocator. Fortunately, that **memcpy** call can be removed and replaced with a single line of code involving no function calls. Examine the five assignment expressions below and circle the one that does the right thing.

```
footer = header;    *footer = *header;    &footer = &header;

header = *(Header **)&footer;        footer = *(void **)&header;
```

d) [5 points] Implement the **find_node** function, which crawls the heap via its implicit free list and, by adopting a first-fit strategy, returns the address of the first free node whose payload is large enough to accommodate the requested size (or **NULL** if there's no such node). You can assume the existence of self-explanatory utility functions with the following prototypes:

```
bool is_free(Header *header);
size_t get_payload_size(Header *header);
```

Use the rest of this page for your implementation.

```
Header *find_node(size_t requested_size) {
   void *curr = segment_start;
   while (curr != segment_end &&
          (!is_free(curr) || get_payload_size(curr) < requested_size)) {
      curr = (char *)curr + get_payload_size(curr) + HEADER_SIZE + FOOTER_SIZE;
   }
   return curr == segment_end ? NULL : curr;
}
```

e) [8 points] You'll be delighted to know you can implement **`mymalloc`** now. I get you started below using code you're already written. It's your job to complete the implementation by marking the selected node as in use and clipping off any excess—provided it's large enough—into a new free node of the appropriate size. For simplicity, you **shouldn't worry about updating the left and right neighbor bits** in any of the nodes. You should, of course, take care to return the address of the freshly allocated payload to the client. You can use **`set_size_and_free_status`** from part b and any other helper functions we've introduced as part of the problem statement.

```
void *mymalloc(size_t requested_size) {
   requested_size = max(MIN_PAYLOAD_SIZE, roundup(requested_size, ALIGNMENT));
   Header *found = find_node(requested_size);
   if (found == NULL) return NULL;
   size_t payload = get_payload_size(found);
   if (payload < requested_size + MIN_NODE_SIZE) { // not enough excess?
      requested_size = payload; // secretly give excess to client
   }
   set_size_and_free_status(found, requested_size, ALLOCATED);
   size_t excess_payload = payload - requested_size;
   if (excess_payload != 0) {
      Header *right =
         (Header *)((char *)found + requested_size + HEADER_SIZE + FOOTER_SIZE);
      set_size_and_free_status(right,
                               excess_payload - HEADER_SIZE - FOOTER_SIZE, FREE);
   }
   return found + 1;
}
```

f) [2 points] Because the design imposes a minimum payload size of 16 bytes, we have the space needed to thread a doubly linked explicit free list through all of the free nodes. Explain how the implementation of **myfree** would be impacted had we opted for a singly linked—that is, next pointers only—explicit free list instead but want to introduce right-coalesce-on-free as your next optimization.

> When the free list is doubly linked, the right node can be spliced out of it in O(1) time, which is crucial to **myfree** if it's also to be O(1). If the free list is only singly linked and forward chaining, your implementation must take O(n) time to walk the free list to identify the predecessor of the right node being coalesced, and O(n) is unacceptably slow if you're to maximum throughput.

g) [2 points] Provided everything about the doubly linked explicit free list is implemented properly, you wonder whether you need to track allocations bits at all. After all, free nodes are in an explicit linked list and allocated nodes aren't. Do the allocation bits provide any value other than convenience?

> They certainly do, for similar reason to those stated in my answer to f. If a node can't self-identify as either free or allocated, it would take O(n) to crawl the free list to see if a node of interest is free or allocated. This would slow down any effort to right-coalesce on **myfree** and in-place **myrealloc** and devastate your throughput scores.

h) [2 points] Assume all three allocation bits—the node's allocation bit and those of its left and right neighbors—are properly maintained, explain why both the left and right allocation bits are useful to the implementation of **myfree**, but only the right allocation bit provides any real value to the implementation of **myrealloc**.

> If all three bits are available, **myfree** can coalesce to both the right and to the left in O(1) time to create as large a node as possible. In principle, **myrealloc** can as well, but in-place **myrealloc** isn't exactly in-place if you have to shift the client payload to the left. So, while **myrealloc** could left-coalesce, there's not much practical value to doing so. (Note that when in-place isn't possible even with right-coalesce, **myrealloc** becomes a **mymalloc**, **memcpy**, and **myfree** of the old node, and in that sense, you could argue that **myrealloc** benefits because **myfree** does.)