

CS107 Heap Allocator Debugging Tips and Strategies

Ben Yan (bbyan@stanford.edu), Winter 2026 Repost

Hello amazing CS107 students! (and soon to be CS107 graduates 🥰🥳)

I hope that you're doing well, and that progress on our glowing capstone and sapphire of the quarter, Heap Allocator, is looking promising :). If you're just starting right now, no worries, that's perfectly ok too!

In any case, I've been wanting to "compile" a guide on tips / strategies for testing and debugging our allocators in GDB, especially when errors manifest on the larger pattern scripts or colossal titan trace scripts. I hope that you find it useful!

Here's a **table of contents for this longass guide** that is shamelessly heap allocator themed (huge thanks to 🙏 former TAs Arman & Andrew, whose stellar guides I'm drawing inspiration from).

- **[Header 1: General Building & Testing Advice]** [60 lines of payload]
- **[Header 2: Upgrading Implicit Into Explicit Allocator Advice]** [30 lines of payload]
- **[Header 3: Writing Dump Heap and Dump Freelist Advice]** [30 lines of payload]
- **[Header 4: Debugging the Small Example Scripts in GDB]** [20 lines of payload]
- **[Header 5: Debugging the Larger Pattern and Trace Scripts]** [too many lines of payload]
- **[Header 6: Tips for Debugging Memory Overwriting - Watchpoints!]** [30 lines of payload]
- **[Header 7: Tips for Debugging Segfaults - Let It Crash in GDB First]** [20 lines of payload]
- **[Header 8: Allocator Performance Metrics]** [10 lines of payload]

✓ Header 1: General Building and Testing Advice

Heap allocator is our largest assignment, so **decomposition and fleshing out your code in digestible bytes and pieces is crucial**. You can do this! **Write abundant helper functions early on instead of later, even if they feel like laughable one-liners and/or busy work and overhead**. I promise this will pay dividends later on and make your life easier. For instance, the logic in setting 8-byte headers (e.g., any casting & dereferencing, bit operations), or in governing the free list of blocks, can be tricky to get right. So instead of having to repeatedly think about and rewrite it, you can write the code once, package it into a helper function(s), and just call that helper to do your bidding. It also consolidates the relevant logic into one place for debugging should an error occur.

Another general point of advice: add features one at a time, and test at each step, instead of trying to implement all features and YOLO testing them all in one go. Like Candy Crush, only move onto the next level / feature when the previous one is done 🎮. When I rebuilt my heap allocators from scratch last Tuesday ("out of the love of the game"), **this was the gameplan I roughly followed for implicit**.

⚠️ **Overarching Note:** Write `dump_heap()` as early as you feel comfortable! You'll need it for GDB debugging. Then, draft your `validate_heap()`, which is similar but also performs checks for issues, e.g., invalid payload size.

1. Write `myinit()` and a basic `mymalloc()` with no splitting of blocks nor handling of edge cases (just search for, find, and allocate a big enough free block to the client). **This should pass the `example1-nofree.script`.**
2. Implement `myfree()`. **This should pass the `example2-recycle.script`**

3. Revise your `mymalloc()` implementation to support block splitting when possible. **This should pass the `example4-coalesce.script`** (don't worry if utilization is cooked ~30%)
4. Implement `myrealloc()`, brushing aside in-place realloc and any efforts at coalescing. **You should pass `example3-inplace.script`**. If bugs occur here, it's worth checking out and stepping through `mymalloc()` and `myfree()` also, as this function relies on them.
5. **Then, run the larger, more diverse pattern scripts.** `pattern-repeat` is a great place to start if it has bugs. For convenience, I like to first debug scripts where the error occurred earlier on :)
6. Next, make sure that your code handles the edge cases in the handout / man pages. **This should pass the `robust.script` slay.**
7. **Lastly, try passing all the `trace` scripts.**

Note we can run a specific script with `./test_implicit samples/[name of script]`, or all of them collectively with `./test_implicit samples/*.script`. You can also read each `.script` file in the `samples` directory.



Header 2: Pokemon Evolution of a Working Implicit Into an Explicit Allocator Advice

When evolving to the explicit allocator, what I did was first copy over all the code, i.e. `cp implicit.c explicit.c`. From there, I followed a series of evolution stages to gradually bring in the shining, mother-pearl features of explicit, e.g., linked list, coalescing, in-place reallocation.

[🔗 Stage 1, Linked List] Switch your code for `myinit()` and `mymalloc()` to construct and support a doubly-linked list of free blocks. In particular, instead of linearly scanning each block on the heap, free or allocated, your `mymalloc()` implementation should iterate exclusively over the list of free blocks inside for a block to allocate. Brainstorm and write any helper functions you think of, and scenarios when blocks should be added/removed from the list. **At this stage, do not modify `myrealloc`, and the only change you'll likely want in `myfree()` is to add the newly freed block to the list.** At this stage, **all scripts with `./test_explicit` should pass; don't worry if utilization doesn't budge.** If any scripts don't pass, I'd recommend following the testing order above for implicit!

[👉 Stage 2, Coalescing] Now, for `myfree()`, implement the "coalescing" of a free block to its physical free neighbor on the right if possible. Decomposing this into a helper function can be nice. **At this stage, all scripts should again pass, and the utilization for `example4-coalesce.script` should improve from the implicit allocator.**

[🌊 Stage 3, In-Place Reallocation] Finally, rehydrate your `myrealloc()` to attempt in-place reallocation, i.e. like a free-flowing river, absorb as many free blocks to the right, and see if we now have enough terrain to satisfy the user's request. If not, we'll need to reallocate elsewhere. My advice: get out your crayons, pencils, any muses of inspiration, and draw out the different scenarios! **At last, all scripts should pass, and the utilization for `example3-inplace.script` should be greatly improved 🎉.**

✓ Header 3: Writing `Dump_Heap()` and `Dump_Freelist()` Early On

Heap allocator is a grand experiment in world-building 🖋️, so very often, we'll need lore dumps of what the world of the heap looks like: the addresses at which headers are placed, how many payload bytes follow each header, whether each block is free or allocated, etc. **This is where `dump_heap()` comes in, which I would recommend writing as soon as possible!** Below is a sample printout from my own boring implementation (Ben "Yawn"), though you're more than welcome to be stylistic and creative ✨.

```
Heap Starting Address: 0x107000000, Ending Address: 0x207000000
Header Address: 0x107000000, Payload Size: 104 Bytes, Status: Free
Header Address: 0x107000070, Payload Size: 200 Bytes, Status: Allocated
Header Address: 0x107000140, Payload Size: 4294966968 Bytes, Status: Free
```



Much of my own debugging experience looked like this: fire up GDB, plant a breakpoint on a relevant request to `mymalloc` / `myrealloc` / etc., and while running the script that failed, **alternate between stepping through sections of code and periodically examining the heap with `dump_heap`, i.e. `(gdb) call dump_heap()`**. This can help you narrow down precisely when the heap state went awry, e.g., a payload size was set incorrectly or accidentally to zero bytes, a block wasn't properly marked as free, a block wasn't split when it should have been, etc.

For the explicit allocator, I would recommend writing a sister function `dump_freelist()`, which, instead of scanning through all blocks linearly starting from the heap start, we instead begin at the head of the free list. We then iterate through the linked list by traversing the pointers, printing out each block's address (and possibly more info) in list order. That way, you can investigate any potential errors with the list, e.g., missing blocks, imposter allocated blocks inside (oh my, eject them), etc. **Alternatively, you can retrofit your `dump_heap()` to print out the `prev` & `next` pointers for each free block (with `printf`, you can use the placeholder `“%p”` for memory addresses in hex).**

✓ Header 4: Debugging with the Smaller Example Scripts (💖GDB is Your Best Friend💖)

The example scripts are amazing in that they evaluate base functionality while staying short and sweet. **I highly recommend reading each of these script files, and drawing out on paper what the heap should look like after each request.** From there, while debugging, you can feasibly break on each `mymalloc`, `myfree`, `myrealloc` request (at most 6 in total), and patiently step through your lines of code to figure out at what point the heap state diverged from what you expected, and diagnose what went wrong. **Setting up the GDB expedition may look something like this.**

```
bbyan@myth: gdb test_implicit (or gdb test_explicit)
(gdb) break mymalloc
(gdb) break myfree
(gdb) break myrealloc
(gdb) run [name of example script]
```



You can also utilize TUI display mode, i.e. `(gdb) layout src`, if you want a majestic dual-panel screen with your code on the top (and the line about to run highlighted), and the GDB console as a regal pedestal below. You may need to hit `(gdb) refresh` occasionally if the screen glitches. And to relinquish this split screen, you can type `(gdb) tui disable`.

✓ **Header 5: Debugging with the Larger Pattern and Trace Scripts (Read & Understand the Test Harness, and Use Conditional Breakpoints)**

Heap allocator has made me cry a few times, and this was the case when my allocator broke down on line 700 of a 1000 line script 🐛😭. This can happen, but trust me, you will catch the bug, as elusive as may seem! But **instead of crawling through every single mymalloc, myfree, etc. call until you reach the one that failed**, which is tedious and probably impossible, **you'll want to leverage the powerful technique of !! conditional breakpoints !!**.

That is, you can set a breakpoint that triggers only when we reach a certain line / request in the script file, using the format `break [line_in_code] if [condition]`. Here, this may appear as:

```
bbyan@myth: gdb test_implicit (or gdb test_explicit)
(gdb) break test_harness.c:181 if (req == [line_in_script] - 1)
(gdb) run [name of script that failed]
```

The requests are 0-indexed, so if we want to skip ahead to line 100 of the script, then we check `if req == 99 to break there`. For each script, you can get line numbers with, e.g., `cat -n samples/[name of script]`. To understand **why we break on line 181 of test_harness.c** (Ben, that's so random lol), **we'll first want to familiarize ourselves with this harness file**. This is the script that's launched into action when you test your allocators. The main attraction is a for loop in `eval_correctness()`, which iterates over each line in the script, checks if it's an 📦 `ALLOC`, 🗑️ `FREE`, or ♻️ `REALLOC` request, and tests the corresponding allocator function.

```
// send each request to the heap allocator and check the resulting behavior
180 for (int req = 0; req < script->num_ops; req++) {
181     int id = script->ops[req].id;
182     size_t requested_size = script->ops[req].size;
183
184     if (script->ops[req].op == ALLOC) {
185         bool fail = false;
186         void *p = eval_malloc(req, requested_size, script, &fail);
```

When you break on line 181 (at the start of the for loop body), you'll want to follow these steps.

1. Run `(gdb) next` through the lines until you encounter a function call of the form `eval_{malloc, realloc, free}`. If you see `myfree()` right away, jump to Step 4.
2. Then, `(gdb) step` to step inside that wrapper function.
3. Keep entering `(gdb) next` until you encounter either `mymalloc()`, `myfree()`, or `myrealloc()`, which is the function call that's wrapped inside and being evaluated.

4. From there, you can perform another (gdb) step, which will transport you into (at last) your own code in `implicit/explicit.c`, allowing you to traverse through and execute lines of your own allocator implementation to see what went wrong, yay!



As an add-on, it's also possible that while the allocator failed on say `req = 700`, that the heap state was **actually compromised earlier**, and the 700th request is merely exposing that internal corruption. For instance, a block could be allocated on `req = 694`, but accidentally not evicted from the free list. This by itself may not kindle an error, but the block may get allocated again on `req = 700`, causing the harness to crash. **The takeaway here is that we may need to work backwards** / place breakpoints in the past to unearth when the heap was truly vandalized.



Header 6: Tips for Debugging Memory Overwriting – Consider Adding Watchpoints!

One of the most common, head-scratching bugs that can occur is when **an existing header's bytes are suddenly engulfed and overwritten with new memory**, without that header being coalesced. I lose a few of my gray hairs every quarter over this, but trust me, we find a way!

One thing to know is that **after allocating a block, our test harness memsets a pattern of bytes over the entire payload**, e.g., a recurring rainbow of `0x2a` or `0x08` in each byte. **In GDB, if you witness this pattern at an address where it shouldn't be there** 🤖 (e.g., when printing out the header, or the prev/next pointers in a free block's payload, you get a hex pattern such as `0x2a2a2a2a2a2a2a2a`), **this suggests these are bytes the harness believes are payload while our allocator believes them to be internal heap data structures, e.g. headers.**



Dramatic irony oh dear. Some possibilities: an allocated block was left stranded on the free list, a block coalescing or splitting was borked, an incorrectly set payload size in the previous header caused the harness to fatefully believe this overwritten header was in the jurisdiction of the previous header's payload.

Sometimes, you can debug this using the same tools as before (e.g., breakpoint on a script request, then stepping through calls to `mymalloc`, etc.), but sometimes the memory overwrite – and mysterious demise of the header into payload – occurs subtly, almost invisibly, and much earlier. Here, **my ultimate late throw of the dice (“with this treasure, I summon...”)** has been the **amazing technique of watchpoints**. Many of my final haunting bugs have been defeated by using this :). **Watchpoints are special breakpoints that only trigger when memory at a specific address is modified (and display both the previous and updated values)**, granting the ability to track down and pinpoint precisely when the bytes were incorrectly overrun, and to rewind to what request / lines of code were likely responsible.



For instance, if we wanted to **monitor a 8-byte header starting from address `0x107000100`**, we could write the below (don't forget to cast & dereference, in the zeitgeist of generics).

```
(gdb) watch *(size_t *)0x107000100
(gdb) run samples/[name of script that erred]
```

If you break on a memory modification that looks innocent, you can **(gdb) continue** to the next one.

Also, note you can print in hex with the `/x` ornament, e.g., `print /x *(size_t*)0x107000100`



Header 7: Tips for Debugging the Notorious Segmentation Fault Errors – Let It Crash in GDB First

A tragically vague error is `Segmentation Fault (core dumped)`. We've probably all had this at some point 😞. **If this occurs when running an allocation script, I would first run it in GDB without any breakpoints!** This may feel strange, but it relies on the fact that when our program crashes in GDB, it conveniently deposits us at where the crash occurred so we can investigate what went amiss.

```
(gdb) run samples/[script that will crash]
... the test_implicit or test_explicit program crashes, leaving us where it crashed ...
(gdb) backtrace
```



Backtrace emits a summary of the 🧊 frozen stack frames that were executing when the program crashed, which may appear as below, if say a line in `mymalloc()` crashed.

```
0. main(argc = 2, argv = ["/test_explicit", "samples/example1-nofree.script"])
1. test_scripts(script_names = ["samples/example1-nofree.script"], num_scripts = 1)
2. eval_correctness(script = "samples/example1-nofree.script")
3. eval_malloc(req = 0, requested_size = 300)
4. mymalloc(requested_size = 300)
```



But it's also very possible the deepest frame that crashed is not `mymalloc()`, but a helper function we called from `mymalloc()` or even the invisible code of a library function you called but didn't write, e.g., a call to `memcpy()`. **My advice for navigating the stack frames is as follows:** (and applies similarly to debugging `myfree` and `myrealloc`)

📌 **If you're in `mymalloc()`**, neat! Inspect the code, call `dump_heap()`, print out the nearby variables, and gather hypotheses on what may have transpired, e.g., dereferencing a `NULL` ptr, invalid memory address.

📌 **If you're deeper in a helper function(s)**, do the same as above with inspecting variables and parameters, but you'll also want to investigate the upper stack frames, e.g., `mymalloc`, to get a vivid sense of how/why those errant parameters were passed in. **You can ascend up a stack frame 📡 by calling `(gdb) up`, and back down with `(gdb) down`. It's like taking the elevator.**

📌 **If you're inside the assembly of a library function, e.g., `memcpy()`, `memmove()`**, you'll have to first **ascend to the nearest stack frame containing your own code**, by calling `(gdb) up` repeatedly, or `(gdb) up [num of frames to ascend]`. Then, investigate what parameters were passed into the library function to cause it to go haywire, and follow the above bullet points.

In any case, I would eventually bubble up to the `eval_correctness()` frame, and print out the `req` variable to get the request number we crashed on. **Once we have this, I'd recommend placing a conditional breakpoint on that request (or earlier), then rerunning the program.** Then, step through the code until you see something go wrong, and voilà, you're getting there!



🧑 **My debugging philosophy: GDB is a bit like GTA** in that if something goes wrong, you can always restart and go back to an earlier (break)point in the game, to analyze & replay the scene differently :)



Header 8: Heap Allocator Performance Metrics

Many students are understandably curious if their allocators, once they're functional, are fast or efficient enough 🤖. (They probably are, but I agree it's worth checking!). And to that end, I wanted to share the utilization averages for my own brand new allocators, while emphasizing that **scores can fluctuate widely depending on the trade-offs you made in your allocator designs. In general, if your utilization average is floating $\geq 50\%$ for both, you're doing wonderful!** 🚢

```
./test_implicit samples/*.script
```

71% average (no coalescing, no in-place realloc,
just the vanilla implicit features)

```
./test_explicit samples/*.script
```

78% average (coalescing, in-place realloc, plus LIFO ordering for free list,
i.e. last-in, first-out)
or 92% average (when altered to sorting free list by memory address 🔥)



There's also the matter of callgrind / instruction counts, which I would look at the Assign6 handout for more details. As one note, if your allocator seems to run much much more slowly when omitting the `-q` flag (i.e. when running `validate_heap()` in between requests to safely-check the heap's internal state), you may want to probe your implementation of `validate_heap`.

Phew, that's all I have for now. If my ultraviolet prose hasn't bored you at this point, tysm for reading all of this! Have an awesome week, and can't wait to see your beautiful allocators!! 🎉