# Malloc Lab Roadmap

## Getting started

Malloc can be intimidating! Before starting, it's important to have a solid understanding of the concepts you'll need for the lab. However, don't spend *too* much time here: you also need to make progress on the lab itself.

1. **Review the concepts.** What does a dynamic memory allocator do? What are fragmentation, throughput, and memory utilization?
2. **Read about "Required Functions" and "Support Routines" in the writeup.** Pay attention to mem_sbrk(), and to the alignment requirement for malloc().
3. **Read through mm.c.** You may find it helpful to print the code out.
    a. Fill in the "What does this function do?" section of each header comment.
    b. Update the .clang-format file to match your preferences for indentation and brace style, and run "make format" to reformat mm.c.
    c. Consider each of the TODOs in mm.c. *Optionally*, answer the TODOs with your own comments. Don't get too hung up on them.
    d. Make liberal use of pre/postconditions, and assert statements for representation invariants. This makes it easy to find out exactly where things have gone wrong.
4. **Write `coalesce_block()`.** This is necessary to make the memory utilization of your implicit list implementation reasonable.
    a. Review the "constant time coalescing" algorithm described in lecture.
    b. Look through the helper functions provided in mm.c, and think about which ones you might need to help you implement coalescing.
    c. Think about the edge cases at the start and the end of the heap. Make sure you don't read data beyond the heap boundaries!
    d. To debug, you can use some of the traces you wrote in the Malloc Lab Traces assignment.

5. **Write `mm_checkheap()`.** This will be graded! Refer to the criteria in the writeup. You should be able to complete most items under "Checking the heap" right away.

    a. Test your checkheap function by running `mdriver-dbg`. The baseline code is very slow, so you might want to individually test a smaller trace with the `-t` flag, rather than running all traces.

    b. Delete a line of code in Case 3 of the `coalesce_block()` function (as long as it still compiles). Run mdriver-dbg again. If your checkheap function does not detect this before segfaulting, *it is not strict enough*.

    c. Use GDB to "track down" the "bug" that you have introduced, once your code crashes. Hmm, a few postconditions could be useful...!

    d. Unfortunately, a strict heap checker can't be invoked in some locations. Instead, make liberal use of assertions (e.g. as a postcondition, assert that a block remains valid once a function returns).

6. **Consider writing a `print_heap()` function** *(optional)*. This will help you track the allocation of blocks over time. It may also help you think of things you're missing in your checkheap.

The rest of this document lists optimizations to mm.c that you might want to implement. Before starting an optimization, think carefully about your goal. Usually, an optimization will help you improve *either* throughput or utilization, but not both.

# Explicit lists

1. **Review the lecture on explicit lists.** Keep in mind that there are *two different meanings of "next block"*: the next block in the heap (implicit list), and the next block in the free list (explicit list).

2. **Update your `block_t` struct** to store any data you need to implement explicit lists. Read the warnings about zero-length arrays, which we do want you to use — but carefully.

3. **Start coding!** Some tips: Reviewing linked-list topics might be helpful. You can adapt linked-list code from elsewhere, as long as you cite it. There are a few design choices that you can make, but keep modularity in mind.

    a. Make a list of every location in the code you'll need to insert a block into the free list, and every location you'll need to remove a block from the free list. Review this list before proceeding.

    b. Checkheap is a debugging tool, so revise it before you start debugging.

    c. Until your code works, focus on correctness over performance. Beware of "optimizations" that could make your code more confusing.

    d. Don't forget to re-initialize *all* global variables every time mm_init() is called.

4. **Experiment with different fit policies** (best fit, next fit, etc.). See if you can come up with a reasonable compromise that will maximize your throughput.

5. **Don't forget to commit regularly.** Ideally, commit each time you get your code working again, with a message explaining what was changed.

6. Once you're done implementing explicit lists, you may be able to get more than 0 points on throughput for the checkpoint. **Make sure you submit to Autolab to assess the performance of your code.** Throughput measurements differ between the Shark machines and Autolab.

## Segregated lists

If your code is modular enough, updating your explicit list to segregated lists should not be a big deal. Some things to keep in mind:

1. If you find yourself copy-pasting code, stop. Think about how you can refactor your code to avoid doing so.

2. Remember that there are *two different meanings of "size"*: the size of the *payload*, and the size of the *block* (including the header and footer).

3. It might be worth performing some analysis of the trace files by modifying mm.c, or by reading the trace files directly, for example with a Python script. How many allocations are under 8 bytes? Under 15 or 16 bytes? Don't waste too much time here, though.

## Eliminating footers in allocated blocks

This is not a difficult task, but it can be a frustrating one if you don't do it carefully. Here's an attack plan to get you through.

1. **Commit your code before beginning.** If you mess up, you'll want a working version of your code that you can revert to.
2. **Make a plan to eliminate footers in allocated blocks.** What *additional data* will you need to store to account for the lack of footers in allocated blocks? Where will you need to change your code to account for this?
3. **Begin storing that *additional data*.** This is the hard part!
    a. Start writing the additional data, but don't actually use it yet. Instead, verify that it is correct wherever you *would* use it, by using assert statements to ensure that it matches the existing data in block footers.
    b. Update your checkheap to verify that the additional data is correct throughout the entire heap. You may also need to relax some earlier invariants. **Please do not skip this.**
4. **Start using your additional data**, now that you know that it's correct. After every step, debug your code until it works. Once it does, commit it.
    a. Stop reading footers of allocated blocks, and use the additional data instead.
    b. Stop writing footers in allocated blocks. If you can do this by changing O(1) lines of code rather than O(n), that might be less error-prone.
    c. Start using the extra payload space you've earned!

# Final

1. Before you start, and as you go, make sure your checkheap function satisfies the requirements listed in the writeup. The list is not exhaustive, but try not to forget any of the items.
2. Consider each of the options in the "Strategic advice" section. How hard would they be, given the current state of your codebase? And how much is each of them likely to improve your utilization?
3. You may want to consult the Malloc Bootcamp for more "strategic advice".

4. Follow the same steps outlined elsewhere in this document: make a plan before you start, update your checkheap as soon as possible, make use of assertions, break your changes into small and testable steps, and commit often.
5. Come see us in office hours if you need help. We *will* ask about your checkheap.
6. Go for it — we believe in you! uwu

# Before submission

1. **Make sure each function has an appropriate function comment**. Do this as you write your code, or you may find yourself doing a lot of typing on the due date.
2. **We expect your header comment to be detailed.** You should briefly discuss, at a minimum: the purpose and layout of your file, the data structures you used, and strategies or optimizations that you implemented.
3. **Think about modularity.** Don't repeat yourself. Make helper functions.
4. When debugging, use `mdriver-dbg`, which uses a lower optimization level, and enables the use of debugging contracts.
5. Don't forget to run `./driver.pl`, which gives deductions beyond those given by `./mdriver` on its own.
6. Verify that your checkheap meets the criteria listed in the writeup.
7. Do a final sweep for magic numbers and dead code.