Assignment 6: Heap Allocator

Due: Thu Dec 8 11:59 pm
No late submissions accepted.

Assignment by Julie Zelenski, with modifications by Nick Troccoli
based on an idea from Randy Bryant & David O'Hallaron (CMU)

Learning Goals

This assignment gives you a chance to implement a core piece of functionality that you've relied on all quarter - a heap allocator! This assignment will help you:

- appreciate the complexity and tradeoffs in implementing a heap allocator
- further develop your pointer and debugging skills
- bring together all of your CS107 skills and knowledge

Overview

You've been using the heap all quarter, and now you get to go under the hood and build your own version of `malloc`, `realloc`, and `free`! The key goals for a heap allocator are:

- correctness: services any combination of well-formed requests
- tight utilization: compact use of memory, low overhead, recycling memory
- fast: services requests quickly

There are a wide variety of designs that can meet these goals, with various tradeoffs to consider. Correctness is of course essential, but is the next priority to conserve space or improve speed? A bump allocator can be crazy-fast but chews through memory with no remorse. Alternatively, an allocator might pursue aggressive recycling and packing to squeeze into a small memory footprint, but execute a lot of instructions to achieve it. An industrial-strength allocator aims to find a sweet spot without sacrificing one goal for the other.

In this assignment, you will implement *two* different heap allocator designs; an **implicit free list allocator** and an **explicit free list allocator**. This assignment leaves room for you to experiment and decide between different possible approaches for implementing these allocators to balance various tradeoffs - beyond the requirements listed in this spec, you are free to design your allocators in the best way you see fit!

To get started on this assignment, clone the starter project using the command

```
git clone /afs/ir/class/cs107/repos/assign6/$USER assign6
```

A note about the Honor Code: An allocator submission is expected to be your original work. Your code should be designed, written, and debugged by you independently. Aside from the provided textbook materials, **all other code is entirely off-limits** -- you must not otherwise study other heap allocator code, incorporate code from outside sources, or discuss code-level specifics with others. A student who is retaking the course may not re-use the design, code, or documentation from any previous effort from a previous quarter.

Getting Started

To get started with the assignment, make sure you've watched everything up through Lecture 23, including the **extra video (https://stanford-pilot.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=f5302931-5e7b-4278-bdd4-af5101485c57)** on explicit allocators and in-place `realloc`, along with our **"Getting Started" videos (https://stanford-pilot.hosted.panopto.com/Panopto/Pages/Sessions/List.aspx#folderID=%2283edf472-7e9a-4df1-8d1d-af51004ffbe3%22)**. They give an overview of the different parts of the assignment, the different files, and how to approach your allocators. They aren't a substitute for reading through the provided code on your own, but we hope they can give you a head start!

View Getting Started Videos (https://stanford-pilot.hosted.panopto.com/Panopto/Pages/Sessions/List.aspx#folderID=%2283edf472-7e9a-4df1-8d1d-af51004ffbe3%22)

View Full Starter File List

Assignment Support/No-Code Helper Hours: Through helper hours, the discussion forum and email, our focus will be on supporting you so that you can track down your own bugs. Please ask us how to best use tools, what strategies to consider, and advice about how to improve your debugging process, but we expect you to take ownership of your own code and drive your own debugging. For this reason, and also because of the wide variety of implementations possible, while we are happy to answer code-level questions, **we will not look at code during helper hours**. This doesn't mean we can't help with code-level questions - it just means we cannot sit with you and look at code together to help diagnose the issue. We can also only answer debugging questions if you have implemented your `validate_heap` and `dump_heap` functions to narrow in on the issue you're having - these functions are a huge asset, and we want to encourage you to take advantage of them!

Code Study: Test Harness

A heap allocator isn't an executable program, but rather a set of functions that can be called from programs. Therefore, to test a heap allocator, we need test programs that call those functions. Our provided `test_harness.c` program is a program that tests calling your heap allocator functions in various ways. It takes as input a test **allocator script** (a text file in a specified format) that it parses to know what requests to send the heap allocator. While executing the script, it attempts to verify that each request was correctly satisfied. It looks for various externally-visible problems (blocks that are unaligned/overlapping/outside segment, failure to preserve payload data, etc). When you compile using `make`, it will create 3 different compiled versions of this program to test with, one using each type of heap allocator: `test_bump`, `test_implicit` and `test_explicit`. You can specify one or more script files as command line arguments to the test harness and it will run and evaluate on each of them. You can also use something called filename *globbing* to specify multiple test files to run with the test harness all at once. For example, you can specify `samples/example*.script` (note the asterisk) as the filename to run all provided test files in the samples folder starting with `example` as its name.

Allocator Scripts

An **allocator script** is a file that contains a sequence of requests in a compact text-based format. The three request types are `a` (allocate) `r` (reallocate) and `f` (free). Each request has an `id-number` that can be referred to in a subsequent realloc or free.

```
a id-number size
r id-number size
f id-number
```

A script file containing:

```
a 0 24
a 1 100
f 0
r 1 300
f 1
```

is converted by the test harness into these calls to your allocator:

```
void *ptr0 = mymalloc(24);
void *ptr1 = mymalloc(100);
myfree(ptr0);
ptr1 = myrealloc(ptr1, 300);
myfree(ptr1);
```

We provide a variety of different test scripts in the `samples` folder, named to indicate the "flavor":

- **example scripts** have workloads targeted to test a particular allocator feature. These scripts are very small (< 10 requests), easily traced, and especially useful for early development and debugging. These are much too small to be useful for performance measurements.
- **pattern scripts** were mechanically constructed from various pattern templates (e.g. 500 mallocs followed by 500 frees or 500 malloc-free pairs). The scripts make enough requests (about 1,000) that they become useful for measuring performance, albeit on a fairly artificial workload.
- **trace scripts** contain real-world workloads capturing by tracing executing programs. These scripts are large (> 5,000 requests), exhibit diverse behaviors, and useful for comprehensive correctness testing. They also give broad measurement of performance you might expect to get "out in the wild". If you have major efficiency issues, these scripts may not complete – if you're noticing these scripts never seem to give you information, try running with the pattern scripts to optimize first!

Your allocator should have no operational or execution errors on any of the provided sample scripts. You are also highly encouraged to create your own!

Test Harness

Read over `test_harness.c` to understand how it operates so you can use it to the fullest. You don't need to dig into the parts that parse a script file towards the end, but do carefully review the earlier code that executes each script and checks correctness.

Verify your understanding by checking in with yourself on the following questions:

- When making a `malloc` request, what checks does the test harness perform to verify the request was properly serviced? What checks does it use on `free`? On `realloc`?
- How does the test harness verify that no memory blocks overlap one another?
- After allocating a block, the test harness memsets a pattern over the entire payload, e.g. repeats 0x2a or 0x08 in every payload byte. (That byte is the request id in the script being processed). If this pattern appears where you think it should not (e.g. a header seems to get randomly overwritten with a repeated pattern, such has 0x2a2a2a2a2a), this means there are bytes that test harness believes is in-use payload yet your allocator thinks those bytes store heap housekeeping. Perhaps an in-use block was incorrectly left on the free list, a block coalesce/split was bungled, or a block is wrongly both in-use and free.
- How does the test harness calculate the allocator's average utilization?
- In addition to its own checks to verify constraints that can be externally validated, the test harness also make calls to `validate_heap`. `validate_heap` is a function that you will write in each of your allocators that does a scan of your internal heap data structures (more about how to implement this later). At what point does the test harness call `validate_heap`? How does the test harness respond when `validate_heap` returns false? What command-line flag configures the test harness to skip the calls to `validate_heap`?
- The test harness calls the allocator's `myinit` function before executing each script. The function is expected to wipe the allocator's slate clean and start fresh. **If `myinit` doesn't do its job, you may encounter bugs caused by ghost values left over from the previous script**. Keep this in mind for when implementing `myinit` in your heap allocator!
- The arguments to `myinit` are the bounds of the heap segment. Run `test_bump` under gdb and print out those boundaries. At what address does the heap segment start? What is the heap segment size? Remember these values, you are going to be seeing a lot of activity in this region.
- Something that may come in handy is breaking at the test harness's execution of a specific line number in a test script. How can we use **conditional breakpoints (/class/archive/cs/cs107/cs107.1232/resources/gdb_refcard.pdf)** to do this? Where might be a good place to add a conditional breakpoint that triggers when we execute line X?

If there is anything you don't understand about the test harness, please ask! You want to have a good understanding of how this code operates so you can use it effectively when testing your allocator.

Code Study: Bump Allocator

The `bump.c` file contains an implementation of a bump allocator. We discussed this strategy in lecture as one of the simplest possible approaches; it is extremely fast, but has very poor memory utilization. The implementation we provide is **not robust to all edge cases and does not perform all required error checking**, but is provided to give you an idea of how you might go about implementing a heap allocator. Here are a few experiments to try out to observe how it performs:

- Read over the C code for the bump allocator. The bump allocator has no per-block memory overhead, but its inability to recycle memory means its utilization is expected to be abysmal. What does the test harness report as the calculated utilization when servicing these allocator scripts?
- Disassemble the `mymalloc` and `myfree` functions and count the number of assembly instructions. This gives you the static instruction count.
- Open the allocator script `samples/pattern-repeat.script` using `emacs` and peruse its contents. The script makes 1000 requests: 500 repeated pairs of a call to `mymalloc` followed by a call to `myfree`.
- Use the callgrind profiler, to be discussed in much more detail in `lab7`, to examine the dynamic instruction count of the bump allocator when running this script:

```
valgrind --tool=callgrind --toggle-collect=mymalloc --toggle-collect=myrealloc --toggle-collect=myfree
./test_bump samples/pattern-repeat.script
```

What is the relationship between the static and dynamic instruction counts? Do you see how they match up?

- Run the annotator, also to be discussed in lab7, on this output file to see the C source labeled with the instruction counts.

```
callgrind_annotate --auto=yes callgrind.out.pid
```

No part of the bump allocator code will stand out as a particular "hot spot". The operations are all pretty streamlined.

- Review the C code for `myrealloc` and its generated assembly. The allocator script `samples/pattern-realloc.script` makes 1000 requests: 100 calls to `mymalloc` calls and 900 calls to `myrealloc` . Use the same profile/annotate steps on this script file. This annotation will report a definitive "hot spot" taking a large number of instructions. What is that time-consuming operation? What makes it so expensive?
- The bump allocator includes a `dump_heap` function that prints out the contents of the heap in text form. But this function isn't called anywhere in the code - so why is it useful? As it turns out, we can call functions while paused in gdb using the `call` command. This means that if we are debugging and want to print out a text representation of the heap contents, we can easily do that with the `dump_heap` function. Check out the `dump_heap` section later in the spec for more information and how to add this to your own allocators.

The bump allocator can service a `malloc` / `free` in just a handful of instructions, but its utilization is extremely poor. In your own allocators, you will aim to strike a balance of good results on both utilization and speed rather than sacrificing one for the other.

Implementing Your Own Allocators

Now it's your turn! Your main task on this assignment is to implement your own implicit and explicit allocators as discussed in class and in the textbook. The B&O textbook chapter 9.9 contains background reading on heap allocators, including sample code. You are allowed to review this code, but note that its structure is somewhat incompatible with our framework. Moreover, its code is not that readable and it uses preprocessor macros (#defines) that behave like functions - **which you should not use**. You can use this code as a starting point, and then write your own. If you do end up taking some inspiration from this code, you may do so given an appropriate citation.

As you work, make sure to set your own target milestones/checkpoints and implement it _in small, manageable chunks, testing thoroughly at each step. We recommend that you do the code studies first to understand how the test harness works, then work through implicit free list implementation. Finally, port everything over to your high-throughput, high-utilization explicit free list implementation - you should start implementing your explicit allocator with your implicit code, which should automatically pass all tests! Then you can focus on adding the explicit allocator features on top of this.

General Requirements

The following requirements apply to both allocators:

- The interface should match the standard libc allocator. Carefully read the `malloc` man page for what constitutes a well-formed request and the required handling to which your allocator must conform. Note there are a few oddballs (`malloc(0)` and the like) to take into account. Ignore esoteric details from the NOTES section of the man page.
- There is no requirement on how to handle improper requests. If a client reallocs a stack address, frees an already freed pointer, overruns past the end of an allocated block, or other such incorrect usage, your response can be anything, including crashing or corrupting the heap. We will not test on these cases.
- An allocated block must be at least as large as requested, but it is not required to be exactly that size. The maximum size request your design must accommodate is specified as a constant in `allocator.h` . If asked for a block larger than the max size, your allocater can return NULL, just as it would for any request that cannot be serviced. You *should not assume the value of this constant beyond that it will not be larger than the max value of a* `size_t` *.*
- Every allocated block must be aligned to an address that is a multiple of the `ALIGNMENT` constant, also in `allocator.h` . *You may assume that this value is 8, and it is fine for your code to work only with an alignment of 8. However, you should still use the constant instead of hardcoding the value.* This alignment applies to payloads that are returned to the client, not to internal heap data such as headers. It's your choice whether small requests are rounded up to some minimum size.
- `myinit` 's job is to properly initialize the allocator's state. It should return `true` if the initialization was successful, or `false` if the parameters are invalid / the allocator is unable to initialize. For the parameters passed in, you may assume that the heap starting address is a non-NULL value aligned to the `ALIGNMENT` constant, and that the heap size is a multiple of `ALIGNMENT` . You should *not* assume anything else about the parameters, such as that the heap size is large enough for the heap allocator to use.
- Your allocator cannot invoke any memory-management functions. By "memory-management" we specifically mean those operations that allocate or deallocate memory, so no calls to `malloc` , `realloc` , `free` , `calloc` , `sbrk` , `brk` , `mmap` , or related variants. The use of other library functions is fine, e.g. `memmove` and `memset` can be used as they do not allocate or deallocate memory.
- **You will need to use a small number of global variables / data, but limited to at most 500 bytes in total**. This restriction dictates the bulk of the heap housekeeping must be stored within the heap segment itself.
- You must have an implemented `validate_heap` function that thoroughly checks the heap data structures and state and returns whether or not any problems were found (see next section).
- You must have an implemented `dump_heap` function that prints out a representation of what the heap currently looks like (see next section).

`validate_heap`

The test harness is written to look for *external* issues/inconsistencies in an allocator, but it cannot peer inside the allocator to check its internal state to make sure it's correct. For instance, the test harness may alert you to the allocator returning NULL, but it may not be able to notice that this was due to the doubly-linked free list getting corrupted a few requests back. To help identify these internal issues, you should implement the `validate_heap` helper function; it is a function that can check for issues with internal heap allocator state. The test harness calls this function periodically, so if something goes wrong you can be alerted at the exact moment it happens. When implementing `validate_heap` , don't repeat the checks the test harness already does for you, but instead augment them by reviewing the internal consistency of the heap. **You should write your implementation of `validate_heap` as you implement each allocator feature - you should not just go back and add it at the end!** As your heap data structures become more complex, `validate_heap` should become more sophisticated to match them.

To provide some examples, your `validate_heap` might walk the entire heap and verify such things as:

- Does the housekeeping information (location, size, free status) for each block appear reasonable? Is all of the heap segment accounted for?
- For your explicit allocator, is every block in the free list marked as free? Is every block marked free listed on the free list? Is each listed only once?
- For your explicit allocator, have adjacent free blocks been coalesced according to your policy?
- Are redundancies in the data structures consistent, e.g. does the count of free blocks match the length of the free list, or the total bytes in-use match the sum of in-use block sizes?

A strong implementation looks for potential issues and reports only on failures. You should not, for example, dump a printout of the entire heap to then manually look through for potential issues. That is something that `dump_heap` (see next section) can do.

Tip 1: We provide a `breakpoint()` function (see `debug_break.h`) that will force a stop in the debugger. You may want to call this function from your `validate_heap` when it detects something is incorrect.

Tip 2: The more comprehensive and thorough checks done by `validate_heap`, the more useful it will be to you. A good `validate_heap` will also be slow -- do not be at all concerned about this, this function is a development/debugging aid, there is no desire for it to be efficient and it will not be invoked during any performance measurements.

`dump_heap`

While debugging, it is very useful to be able to see what the layout of the heap (e.g. free blocks, allocated blocks, etc.) looks like at any given point in time. The `dump_heap` function is a helper function that should do this - it is a function that is not called anywhere in the code, but is useful to call from within GDB (using `call`) while you are working to print out the current contents of your heap. Like `validate_heap`, it is an extremely useful tool to gather information, distinguish between a valid and invalid heap configuration and pinpoint the exact moment the heap got out of whack. The bump allocator provides an example dump heap implementation - for your allocators, you should implement your own dump heap function that, for instance, traverses the heap and prints out information about each block (e.g. info in its header, where the next/previous free blocks are in your explicit allocator, etc.), one block per line. This can show you what your heap structure looks like overall, which blocks are allocated vs. free, what size different blocks are, etc.

1) Implement An Implicit Free List Allocator

Now you're ready to implement your first heap allocator design. The specific features that your implicit free list allocator must support are:

- Headers that track block information (size, status in-use or free) - you must use the header design mentioned in lecture that is 8 bytes big, using any of the least significant 3 bits to store the status
- Free blocks that are recycled and reused for subsequent malloc requests if possible
- A malloc implementation that searches the heap for free blocks via an implicit list (i.e. traverses block-by-block).

Your implicit free list allocator is **not** required to:

- implement any coalescing of freed blocks (the explicit allocator will do this)
- support in-place realloc (the explicit allocator will do this); for realloc requests you may satisfy the request by moving the memory to another location
- use a footer, as done in the textbook
- resize the heap when out of memory. If your allocator runs out of memory, it should indicate that to the client.

This allocator won't be that zippy of a performer but recycling freed nodes should certainly improve utilization over the bump allocator.

The bulleted list above indicates the minimum specification that you are required to implement. Further details are intentionally unspecified as we leave these design decisions up to you; this means it is your choice whether you search using first-fit/next-fit/best-fit, and so on. It can be fun to play around with these decisions to see the various effects. Any reasonable and justified choices will be acceptable for grading purposes.

We strongly recommend approaching this incrementally: for instance, start just by implementing `myinit` and a basic `mymalloc`, and then test them. Then add `myfree`, and later `myrealloc`. You can (and should) test and develop incrementally!

2) Implement An Explicit Free List Allocator

Building on what you learned from writing the implicit version, use the file `explicit.c` to develop a high-performance allocator that adds an explicit free list as specified in class and in the textbook and includes support for coalescing and in-place realloc.

The specific features that your explicit free list allocator must support:

- Block headers and recycling freed nodes as specified in the implicit implementation (you can copy from your implicit version)
- An explicit free list managed as a doubly-linked list, using the first 16 bytes of each free block's payload for next/prev pointers
  - Note that the header should **not** be enlarged to add fields for the pointers. Since the list only consists of free blocks, the economical approach is to store those pointers in the otherwise unused payload.
- Malloc should search the explicit list of free blocks
- A freed block should be coalesced with its neighbor block **to the right** if it is also free. Coalescing a pair of blocks must operate in O(1) time. You should perform this coalescing when a block is freed.
- Realloc should resize a block in-place whenever possible, e.g. if the client is resizing to a smaller size, or neighboring block(s) to its right are free and can be absorbed. *Even if an in-place realloc is not possible, you should still absorb adjacent free blocks as much as possible until you either can realloc in place, or can no longer absorb and must reallocate elsewhere.*

Your explicit free list allocator is **not** required to (but may optionally):

- coalesce free blocks **to the left** or otherwise merge blocks to the **left**
- coalesce more than once for a free block. In other words, if you free a block and there are multiple free blocks directly to its right, you are only required to coalesce once for the first two. However, for realloc you should support in place realloc which may need to absorb multiple blocks to its right.
- resize the heap when out of memory. If your allocator runs out of memory, it should indicate that to the client.

This allocator should result in further small gains in utilization and a big boost in speed over the implicit version. You will end up with a pretty snappy allocator that is also strong on utilization -- an impressive accomplishment!

The bulleted list above indicates the minimum specification that you are required to implement. Further details are intentionally unspecified as we leave these design decisions up to you; this means it is your choice whether you search using first-fit/next-fit/best-fit, and so on. It can be fun to play around with these decisions to see the various effects. Any reasonable and justified choices will be acceptable for grading purposes.

Performance and Efficiency

Note: do not use Valgrind memtool (the memory tool we have used up to this point) on this assignment. It looks only at the standard heap allocator, not your custom one, and its complaints may needlessly worry you.

Utilization

You can achieve higher utilization by more effectively packing blocks into the heap segment so as to accommodate more payload data and correspondingly have less overhead/fragmentation. The test harness computes the utilization per-script and reports the average utilization over a set of scripts. The per-script utilization is calculated as the ratio of the peak payload divided by the total segment space used to accommodate that payload data. Utilization ranges from 0% to 100%. The achieved utilization depends on the script; a certain tough script might have utilization down near 30%, while another script might achieve a near perfect packing of 95%. If utilization averages out at > 50% overall, you're doing great!

Performance

What are some of the expectations about the performance for the different allocators?

Bump. Given it has no variation in the work it does to service a request, both malloc and free are O(1) for bump. We clocked the bump allocator processing each malloc in about 35 instructions at -O0 (and reduced to just under 10 when compiled -O2, wow!). These counts are untouchably speedy, but the consequence of its fast service is extremely poor utilization.

Implicit free list. To service a malloc request, this allocator searches the implicit list. Instruction counts will vary depending on how full the heap is and where the free blocks are found. In the best case scenario (free block found right away), a malloc request can be serviced in about 40 instructions. In the worst-case (search most/all of heap to find block), the count rises linearly relative to the number of blocks in heap. Searching a heap of 1000 blocks can take many thousands of instructions -- ouch! free is a quick constant-time operation, accounting for about 10 instructions per call. The implicit free list can achieve average utilization in the most satisfying 60% range over the pattern/trace scripts, as these workloads benefit from the recycling that implicit does. Hitting above 50% utilization is a very respectable achievement for any allocator!

Explicit free list. In its best-case scenario (free block found right away), malloc can be about 50 instructions. In the worst-case, search bogs down to O(N), but this is linear in the number of *free blocks*, not the total number of blocks. Coalescing a pair of blocks should be O(1), likely on the order of about 30 instructions. This work may be counted in your free/malloc/realloc depending on your design. For some workloads, the more aggressive recycling will lead to even higher utilization for explicit than implicit, but on average, it will be in a comparable range.

Realloc is a slippery operation to measure. If a block is being resized in place, it can be fast to service (about 30 instructions), but otherwise it incurs sum of counts for malloc+memcpy+free. Copying the payload data likely dominates the total cost, so the count of instructions varies based on block size.

The numbers above give you a general sense of what is possible, but are not intended as absolute requirements. These counts were taken from a simple implementation of the minimum requirements with little effort at optimizing other than compiling at `-O2`. Your instruction counts might be a bit higher, and that's fine. If you work to get your counts lower, even better!

Note: Rather than get hung up on absolute numbers, focus your attention instead on what you can learn from relative comparisons. For example, comparing the performance of two different allocators on the same workload allows you to observe the contrast between two approaches. Comparing an allocator to itself before/after a design/code change allows you to measure the impact of that change. Comparing an allocator to itself on a variety of workloads shows its strengths/weakness in different scenarios.

Compiler Optimizations

Once you have completed and thoroughly tested your implementation, feel free to change the top part of the Makefile to tell `gcc` to more aggressively optimize your code. While still in active development, using `-O0` generates code that is much easier to debug, but once you're working on performance, experiment with more aggressive levels and options, such as `-O2`. We will compile your allocator using the setting from your submitted Makefile, so be sure to submit with the flags configured to work best for your code. Remember, however, that optimizing transformations make the assembly much less of a literal rendition of the C, so stepping through it may appear to jump through the code erratically and show unexpected results. **NOTE: `make` doesn't detect changes to the Makefile itself, so after changing the Makefile, run `make clean` and then `make` to recompile.**

Code Quality

- **Write helper functions.** There will be a few vital expressions that are heavily-repeated, such as going back and forth between a payload pointer and its header or advancing from one header to the next. Even though these expressions are one-liners, they are complicated, dense, pointer-laden one-liners. Move these expressions into tiny shared helpers so you can write them correctly once!
- **Pointee types should communicate the truth.** If you know the type of a pointer, declare it with its specific type. Use `void*` only if the pointee type is not known or can vary. Conversely, don't declare what should be a `void*` pointer with a specific pointee type, as this misleads the reader and compiler into thinking it's something that it's not.
- **Structs and typedefs are your friend.** defining structs for groups of data can help make your code cleaner and easier to read. Plus, you can cast to a struct to make it easier to manipulate data. Moreover, you can use `typedef` by itself to declare a custom variable type that is really something else, e.g. `typdef int custom_type;`, to declare `custom_type` that is really an `int`. Here's an example of where a struct might be useful; say we have the following struct:

```
typedef struct mystruct {
    int x;
    long l;
    char *str;
} mystruct;
```

Let's say we had a pointer to memory that stores an `int`, `long` and `char *` together, just like this struct, and we wanted to change one of the fields. Without structs it might look something like this:

```
void *ptr = .... // pointer to the int, which is followed by a long, and char *
*(long *)((char *)ptr + sizeof(int)) = 2; // change long
*(int *)ptr = 1; // change int
*(char **)((char *)ptr + sizeof(int) + sizeof(long)) = .... // change str
```

If we instead made a struct pointer pointing to this area of memory, it is much easier:

```
mystruct *ptr = .... // pointer to the int, which is followed by a long, and char *
ptr->l = 2; // change long
ptr->x = 1; // change int
ptr->str = ... // change str
```

Having a struct lets us much more easily access each individual field because the struct definition declares what data is stored and in what order.

- **Be wary of typecasts.** Typecasts are not to be thrown about indiscriminately. Your typecast defeats the type system and coerces the compiler to go along with it without complaint. Before introducing one, consider: why is this cast necessary? what happens if it is not present? is there a safer way to communicate my intention? how might I work within the type system rather than against it?
- **Decompose for the win**. A page-long malloc or realloc can be overwhelming. Break it down into tasks: finding a block, updating the housekeeping, etc.
- **Names matter**. Choose good names for your fields and functions. A vague name such as `size` forces you to repeatedly ponder whether it tracks the payload size or total block size (included the header), and what units it uses (bytes? etc.). Always use descriptive variable names.
- **Avoid co-mingling of signed/unsigned types.** The standard malloc interface takes its arguments as a `size_t`, which is an `unsigned long`. Careless co-mingling of signed and unsigned types and being sloppy about bitwidth can lead to unexpected troubles (as you saw in the recent assignment), so be thoughtful and consistent.
- **Coding for efficiency**. Our grading priorities typically de-emphasize efficiency. We designate a few points for meeting the "reasonable efficiency" benchmark and focus more on hacky algorithms, code duplication, lack of decomposition, unnecessary special cases, and other choices that make for ugly code. The priorities shift somewhat when efficiency is in play. The ideal is streamlined code that is also clean and high-quality, but should you have to sacrifice a bit on quality in the name of performance, only do so if the performance gain you're getting is measurably important and there is no elegant way to get the same boost.

Readme

Answer the following two questions in your readme:

1. For one of your allocators (either implicit or explicit), address the following in 1-2 paragraphs:
    - describe **two** design decisions you made for the allocator and why you made those choices. E.g. how did you design and represent your header and payload? How do you search for free blocks? etc.
    - Provide **one** example of a scenario (e.g. a specific pattern of requests, a specific size of request, etc.) where your allocator would show strong performance, and **another** example of a scenario where your allocator would show weaker performance, and explain why.
    - Also describe **one** attempt you made at optimizing your allocator *other than changing the compiler flags*.
2. For this assignment, we mention that we will not test with improper requests, and there is no requirement on how to handle improper requests. However, improper heap requests are a common source of bugs (and by this point, everyone in CS107 has likely had a few errors due to misuse of heap memory!). They are also **a common source of vulnerabilities (https://securitylab.github.com/research/CVE-2020-6449-exploit-chrome-uaf/)**, as we saw in the lecture on `malloc` / `realloc` / `free`. For one of your allocators (either implicit or explicit), pick **two** assumptions that your heap allocator makes about the client's behavior and, with your knowledge of your allocator, explain what would happen if the client violated those assumptions. E.g. what would be overwritten? Would your allocator crash? etc. Your assumptions should be about `mymalloc`, `myrealloc`, or `myfree` (and not `myinit`).

As you finish your readme, we have one final question to ask you:

What would you like to tell us about your quarter in CS107? Of what are you particularly proud: the effort you put into the quarter? the results you obtained? the process you followed? what you learned along the way? Tell us about it! We are not scoring this question, but wanted to give you a chance to share your closing thoughts before you move on to the next big thing.

Sanity Check

Custom sanity check is configured to gather instruction count statistics. It runs the allocator under callgrind—you'll learn more about that in **lab7 (/class/archive/cs/cs107/cs107.1232/lab7)**—to get the total instruction count and divides by the number of requests to report the average instruction count per request. In your `custom_tests` file, each line should be an allocator test on a script:

```
test_implicit samples/pattern-recycle.script
test_explicit samples/pattern-recycle.script
```

Running `tools/sanitycheck custom_tests` will report the average instruction count for each such test listed in your custom_tests file.

Submitting

Once you are finished working and have saved all your changes, check out the guide to **working on assignments (/class/archive/cs/cs107/cs107.1232/working-on-assignments)** for how to submit your work. We recommend you do a trial submit in advance of the deadline to allow time to work through any snags. You may submit as many times as you would like; we will grade the latest submission. Submitting a stable but unpolished/unfinished version is like an insurance policy. If the unexpected happens and you miss the deadline to submit your final version, this previous submit will earn points. Without a submission, we cannot grade your work.

Grading

Here is the tentative point breakdown for this assignment:

Functionality (122 points)

- **Sanity/sample scripts** (36 points) Correct results on the default sanity check tests and published sample scripts. (18 points per allocator)
- **Comprehensive/performance** (42 points) Fulfillment of the specific implicit/explicit requirements and correct servicing of additional sequences of mixed requests. The performance expectations are fairly modest and intended to verify that the allocator correctly implements the required features. (20 points for implicit, 22 for explicit)
- **Robustness** (22 points) Handling of required unusual/edge conditions (request too large, malloc 0 bytes, etc.) (10 points for implicit, 12 points for explicit)
- **Clean compile** (2 points) We expect your code to compile cleanly without warnings.

- **readme.txt**. (10 points) The readme questions will be graded on the thoughtfulness and completeness of your answers. (5 points per response)
- `validate_heap` (7 points) We will evaluate the usefulness and thoroughness of your validate heap function
- `dump_heap` (3 points) Your dump heap function should print out a useful textual representation of the heap, and not just be a copy of the provided `dump_heap`

Code quality (buckets weighted to contribute roughly 20 points)

The grader's code review is scored into a bucket per assignment part (implicit, explicit, validate heap) to emphasize the qualitative features of the review over the quantitative. The **styleguide (/class/archive/cs/cs107/cs107.1232/styleguide)** is a great overall resource for good program style. Here are some highlights for this assignment:

- We expect your allocator code to be clean and readable. We will look for descriptive names, helpful comments, and consistent layout. We expect your code to show thoughtful design and appropriate decomposition. Control flow should be clear and direct. We expect common code to be factored out and unified. Complex tasks or tricky expressions that are repeated should be decomposed into shared helpers. See the style guide on the assignments page for more information.
- The code review will also assess the thoroughness and quality of your `validate_heap` routines, and whether you implemented a `dump_heap` function.

Post-Assignment Check-in

How did the assignment go for you? We encourage you to take a moment to reflect on how far you've come and what new knowledge and skills you have to take forward. Once you finish this assignment, you will have put all your hard work this quarter in CS107 towards implementing a crucial tool in program execution, and one that at the start of the quarter you took for granted. Completing this assignment requires use of pointer skills, memory manipulation, assembly optimization, and more - an amazing accomplishment. It's definitely time to celebrate! **Congratulations on an awesome quarter!**

To help you gauge your progress, for each assignment/lab, we identify some of its takeaways and offer a few thought questions you can use as a self-check on your post-task understanding. If you find the responses don't come easily, it may be a sign a little extra review is warranted. These questions are not to be handed in or graded. You're encouraged to freely discuss these with your peers and course staff to solidify any gaps in you understanding before moving on from a task. They could also be useful as review before the exams.

- What are the main challenges in making malloc fast? What makes for a fast free?
- Explain the difference between internal and external fragmentation. Which is the greater threat to utilization?
- True or False: If you have built a super-fast malloc and free, implementing realloc in terms of those operations will also be super-fast.
- Throughput and utilization are often viewed in opposition. How might improved throughput come at the expense of lowered utilization (or vice versa). Are there choices that lead to wins in both?