Common Problems with free

Our locker analogy made references to two common errors that programmers make when using **malloc** and **free**. Trying to "free" the same lockers twice is a problem. In the case of memory allocation, trying to free the same block of memory more than one time is called *double freeing*. Generally, your program will crash (*segfault*), although, other more sinister behaviors can occur. A segfault on the line where the double free happened is nice: it makes debugging easier. However, you may get stranger symptoms—including your program crashing the next time you call malloc. In general, if malloc crashes, an earlier error in your code has corrupted its bookkeeping structures, and you have just now exposed the problem. Run your code in *valgrind*, and it is quite likely to help you expose the error sooner.

Another common problem, also alluded to in the locker analogy, is freeing something that is not at the start of the block returned by malloc. If the locker attendant gave you lockers 37–41, you cannot go back and say "I'm done with 38." This may seem silly: why can't the locker attendant just figure out that when you say you are done with 38, you mean the block from 37–41? For a human tracking lockers, this may seem like a silly rule; however, it makes much more sense for malloc and free.

Neither of these functions is magical (nothing in your computer is—as you should have learned by now). They need to do their own bookkeeping to track which parts of memory are free and which are in use, as well as how big each block that is in use is. Bookkeeping requires memory: they must store their own data structures to track the information—but where do they get the memory to track what memory is in use? The answer is that malloc actually allocates more memory than you ask for, and keeps a bit for itself, right before the start of what it gives you. You might ask for 16 bytes, andmalloc gives you 32—the first 16 contain its information about the block, and the next 16 it gives you to use. When you free the block, the free function calculates the address of the metadata from the pointer you give it (*e.g.*, subtract 16). If you give it a pointer in the middle of the block, it looks for the metadata in the wrong place.

Going back to the locker analogy, this would be as if the locker attendant gives you lockers 37–41, but then puts a note in locker 36 that says "this block of lockers is 5 long." When you return locker 37, he looks in locker 36 and finds the note. If you instead tried to give back locker 38, he would look in locker 37 and become very confused.

A third common mistake is freeing memory that is not on the heap. If you try to freea variable that is on the stack (or global), something bad will happen—most likely, your program will crash.