Our design of a work stealing threadpool is largely based around three structures: `future`, `thread`, and `thread_pool`.

`future` is the simplest of the three structures. It contains an enum that represents the state of the future (either not started, in progress, or done), the future function itself, and two pointers to the data and the result buffers respectively. Additionally, it contains a semaphore that the process polling the future can wait on if the future in is in progress when it is polled. Finally, it contains a pointer to the `thread` whose queue it may be in alongside a pointer to the `thread_pool` it belongs to so that functions operating on it have easy access to them.

`thread` is probably the most crucial structure. There's decently heavy contention over the thread objects, so properly optimizing them was paramount for good performance. The most important part of `thread` is the list of jobs and the lock protecting it. The list contains all jobs spawned by the thread that have yet to be executed by any threads. `thread` also contains its thread number, which is used for an important performance optimization. Finally, it contains its thread id and a pointer to the `thread_pool` for convenience reasons. `thread` is also aligned to 64 bytes to avoid issues with different locks on the same cache line.

`thread_pool` is the glue that holds everything else together. It hold the global queue and its lock, the total number of threads, a condition variable to signal new tasks, and the shutdown bool. Additionally, it contains an array of `thread`s as well as a dummy `thread` that is used when the main thread needs to perform computations.

Most of our function designs come directly from these structures. `thread_pool_new` initializes `thread_pool` and spawns the appropriate amount of pthreads while initializing their `thread` structs.
`thread_pool_shutdown_and_destroy` sets the shutdown bool, broadcasts to any thread that may be for a new task, reaps each thread, and frees the few mallocs we did. `thread_pool_submit` checks if the `future` is being added by the main thread or a spawned thread. If it's the latter, it adds the future to the thread's queue. Otherwise, it adds the future to the global queue.

`future_get` is a bit more complex. First, it checks the status of the `future`. If the `future` is in progress or done, it waits until the semaphore in the `future` is incremented then returns the data. If the `future` has not been started, it finds the `future` in in whatever queue contains it and executes it if it has still yet to be started. If the `future` was started in the remaining time, it waits on the semaphore.

Finally, we have the various functions for the worker threads themselves. The base function is `worker thread`, which loops around checking for work. It first checks if there is any work in its queue. If not, it checks for work in the global queue. Then, it calls a helper function `steal_work` that checks if there is any work to steal from one of the other threads. Finally, it checks if it's time to shut down, and if not it waits on the condition variable for more work. If at any time in this process the thread finds work to do, it calls `do_work` with the work and then goes back to the top.

`steal_work` iterates through each `thread` and checks if any thread has work available to steal. If so, it steals the work and returns it to `worker_thread` to execute. `steal_work` checks in a circular fashion. That is, if it is thread k and there are n threads, it checks $k+1, k+2 \ldots, n-1, n, 1, 2, \ldots, k-1$. This massively reduces contention by starting each thread at a different point. That way, if many threads are trying to steal work at the same time (e.g. if several threads just got off of waiting on the condition variable), they don't all queue one by one for each lock.

`do_work` is a simple function. It sets the thread's state as in progress, exectutes the work, then sets the thread's state to finished and increments the semaphore.