Here is the template for creating a new thread:

```
-------------------------------------------------------
int pthread_create(
pthread_t *thr              // contains new thread's ID
const pthread_attr_t *attr   // give attributes
void *(*start_routine)(void) // name of function that the thread will run.
                        //  Must have void pointer as its return and
                        //  parameter values
void *arg                  // void pointer for passing parameters
------------------------------------------------------
```

Note that there is only one place to pass arguments to the thread, the void pointer that is the last paramter of the function. This turns out to be a completely general way to pass parameters, but a perhaps a little non-obvious/unusual. Some of the following is taken from http://www.faqs.org/docs/learnc/x658.html. First about void pointers themselves.

When a variable is declared as being a pointer to type void it is known as a generic pointer. Since you cannot have a variable of type void, the pointer will not point to any data and therefore cannot be dereferenced. That is, the compiler won't let you get a value from the pointed to location (using "*") because it doesn't know how many bytes to get. A void ptr is still a pointer though, but to use it you just have to cast it to another kind of pointer first. Hence the term generic pointer.

A generic pointer is very useful when you want a pointer to point to data of different types at different times, as in when you are creating a generic thread.

----
test_generic.c

test_generic.c uses a void pointer (without threads). Note that the cast (int*) makes the type of pointer into an "int" type. The *(int*) makes the variable into a pointer. The way to read this is out-to-in. Once you get this basic structure, you can pass anything, as long as it has a defined starting address. (You can even make the address itself the argument to pass by value, as is shown below.)

Task 1. Modify test_generic.c so that it used the generic pointer to print out the three elements of "float ArrayA". You should be able to do this in two ways: by adding the offsets (0,1,2) to the pointer and by using array syntax [0],[1],[2]. Hint: be careful with your parentheses!

----
test_create.c

This is the "Hello World!" of thread creation. Read and understand the code. In this example of pthread_create() we are not declaring any parameters or special attributes; therefore the values for these fields are NULL. Even so, a pointer is passed to the created function so that it knows where to store the thread ID value it generates. It also passes the work function, which denotes where the new thread will begin.

Task 2. Compile and run the code. Record the output. Try it a few times. Does the output change? (It may or may not!)

Task 3. Modify test_create.c so that "id" is declared using malloc, and pass the location to store the thread ID information in pointer form (instead of a referenced array).

Task 4. Add a sleep(3); line to the work() function before the printf statement and compile and run. Observe the output when the program is run. What do you see? Why?

Task 5. Remove sleep(3) from work() and add it to main() between the printf() and return(). Compile, run, and observe the output. What do you see. Why?

What you are seeing is the main thread completing before the child threads can reach the printf statement. When the main thread completes, all child threads are automatically killed.

----
test_join.c (similar to ex3 in the class notes)

This time after the threads are created, main() walks through the list of thread Ids with the pthread_join function, and will successively block on each until that child thread has completed.

Task 6. Add the sleep(3); to the child process again, and observe the output. Does it change? Why or why not?

Keep this in mind, because you may need to add joins to the remaining examples to get the expected results.

----
test_param1.c (similar to ex2b.c in the class notes)

Now let's pass a value to the new thread. Look at test_param1.c. We are passing the value of t by casting t to a generic pointer. NOTE: We are not passing the memory location of t, then fetching the value at that location in the new thread. Rather, we are telling the compiler that the value of t is actually a generic pointer, and then the new thread is telling the compiler that no, it is not a memory location, it is actually a value. As long as the type that gets converted to the generic pointer is the same as that to which the generic pointer is converted, this should work.

Task 7. To prove this statement, print the value of threadid before it is cast back to a "long unsigned". It should be the same number. Now try passing something really different, like a signed char set to some negative number. Does it still work (or compile)? Why or why not?

Task 8. Also, what's happening on the print out? Does it change if you run it again? Why?

----
test_param2.c (similar to ex2a.c in the class notes)

Look at test_param2.c, and observe that you can also pass by reference (in addition to the unorthodox "pass by value" shown in test_param1.c). (In test_param3.c, the method here is used to pass a pointer to an entire structure to a thread.) Note that the threads cast the generic pointer (now the input) in two ways: as an integer and as a pointer to an integer. When you change the value being pointed to, this change is visible to all other threads.

Task 9. In test_param2.c, confirm that threads affect each others values. Modify f before the printf, compile and run. Now do the same with *g. What happens? Why? Does your output change from run to run? Without changing the loop structure of the thread creation loop in main(), try to modify your code so that fewer than NUM_THREADS gets created.

Task 10. Modify test_param2.c to pass an int array (the same one) to each thread. Show that each thread is accessing the same array by having each thread modify a different element, and then printing out the final array. How do you get each thread to know which number it is (without using explicit synchronization)? Will your method always work?

----
test_param3.c (an extension of ex2c.c in the class notes)

Since pthread_create has only one parameter slot, the only way to pass more than one value is by reference. In test_param2.c, all threads were manipulating the same int and then the same int array. Also, it was hard for them even to tell who they were themselves. (Of course you can look at your own ID, but this doesn't tell you anything about how you relate to the other threads.)

Observe that test_param3.c passes complex parameters by reference, but also that each one is tailored to its particular thread, with its own ID assigned by main().

Task 11. Compile and run test_param3.c to verify this behavior. Now add a 6th thread whose "sum" field contains 1000.

----
test_sync1.c

So far we have "sync'ed" in one way: by joining threads before exiting. This prevented the race condition caused by the master exiting before the child threads. In the last few sections we also amused ourselves by causing various outputs due to having multiple threads modify shared data. In serious programs this is seldom desired. Much more often we want to control when threads execute code which calls or modifies shared data. This can be done using mutexes.

test_sync1.c spawns a thread which cannot reach its print statement until the main thread releases control of the mutex. This is done by entering a character and pressing return.

Task 12. Comment out the trylock check in the child thread and observe the difference in output.

Task 13. Now change NUM_THREADS to 2 (and change the *Messages initialization), compile and run. Does it work?

----
test_barrier.c

In #13 you observed that sync'ing multiple threads using a single mutex is problematic.

Now look at test_barrier.c. When a thread reaches a barrier call, it will block until the number of threads specified in the barrier declaration have hit the barrier, at which point all threads will unblock.

Task 14. Compile and run test_barrier.c. Observe that all of the "before barrier" messages are printed before the "after barrier" messages. Modify test_barrier.c by uncommenting the "sleep(1)" statement. Observed the change in behavior. Does the barrier still work? Change to sleep time to a variable function of taskid. Does the barrier still work?

----

test_sync2.c

Sometimes barriers are too clumsy. That is, we would like threads to execute in a particular sequence. See, e.g., the graph in the notes, test_sync2.c implements this graph:

2 & 3	wait for 1 (main())
4	waits for 2
5	waits for 4
6	waits for 3 & 4
7	waits for 5 & 6

The method used is for main() (thread 1) to create and lock all of the mutex locks and then create threads 2-7. These threads look at their IDs (passed from main()) and go the the corresponding "case." There they wait for the specified lock (2&3 for 1, etc.). When this thread is unlocked by another thread, this thread continues (break;). At the end of the thread, it unlocks a lock corresponding to its own ID. This causes other threads to be released. The cascade begins when main() unlocks mutex 1.

Note that test_sync2.c uses LOCK to wait for an UNLOCK rather than TRY. This blocks and is often more efficient.

Task 15. Compile and run test_sync2.c and confirm this behavior. Note that the join() loop is executed after most of the threads have exited. Why does this work?

Task 16. Modify test_sync2.c to add a thread 8. It waits for threads 6&7.

----

test_crit.c

The critical section problem arises when there is a section of code whose simultaneous execution by multiple threads causes an error. test_crit.c has the classic example: some threads add to the balance, some subtract. In the end, the balance should stay the same.

Task 17. Compile and run test_crit.c. Is the answer correct? That is, is the final balance the same as the beginning? Perhaps you find that it is! (So what's the problem?) Can you change the timing, but not the logic, so that the balance becomes incorrect? How about if you increase the number of threads, to say 10,000?

Task 18. Use mutex to solve this problem.