Data Structures

Assignment 4

(1) (50 pts) Write a stack class. It should support push, pop, is_empty, and clear operations. You do not have to program the dynamic memory allocation. You may use a vector to store the values and have your interface map to vector operations. You may add any additional helper or debugging functions you may need.

Write a class representing a math co-processor. It should have a stack memory. It should support the following instructions:

push x	push x to top of stack
pop	return top of stack and remove it
peek	return top of stack but don't remove it
+	remove top two elments and replace with their sum
-	remove top two elements and replace with their difference
*	remove top two elments and replace with their product
/	remove top two elements and replace with their quotient and remainder
sin	replace top of stack with the sin of the value
sincos	replace top two elements of stack with the sin and cos of the values
sqrt	replace the top of the stack with its square root

Write driver code in main to show these functions working. Also write a sequence of these instructions that implements the quadratic formula and demonstrate it working.

You may process instructions from a vector (or array) of strings. The co-processor should have an execute function that takes a string parameter, parses the instruction, and then updates its stack accordingly. Have a private method for each of the instructions and call the appropriate one from the execute function.

(2) (50 pts) Write a priority queue class. It should support push, pop, is_empty, and clear operations. As with the stack class, you do not have to program the dynamic memory allocation directly. You may use a vector to store the values and map your interface to vector operations. You'll need to figure out a way to represent the priority along with a value. You may add any additonal helper or debugging functions, or even other classes, as your design may require. You should write this so that the lower priority number is the higher priority level.

Write an interrupt handler class. It should resolve interrupts according to their priority. Use your priority queue class as the interrupt handler's memory. It should have a poll function that takes a string with an interrupt name and an int with a priority level. It should have a process function that processes the highest priority interrupt in its queue. Just have this function output the interrupt name that's it's processing and remove that entry from the queue.

Send the following list of interrupts to the handler and demonstrate they are processed in priority order:

Interrupt	**Priority**
timer1	2
timer2	4
serial	3
none	
external1	2
timer1	5
none	
external2	3
serial	1
timer0	3
none	
external1	4
serial	2

Call the poll function twice with the above interrupt info, then call the process function, then call the poll function twice, then process, etc. When the list ends, finish with calls to process until the queue is empty.