The purpose of this assignment is to give you practice implementing classes with dynamic memory, implementing a simple interactive command processor, using `get`, and using an enumerated type. In addition, it will give you more practice with pointers and heap memory in general and with implementing classes.

For this assignment you are to write a program which provides a simplified simulation of a one worker copy center for a company. The company has a number of departments, so copy jobs are always labeled with the department number. Each job has a specified number of copies to make. In addition, for the day, each copy job is given a unique id, which is provided by the program (not the input). Each job has a priority (which is in the input), which is simply an integer – the higher the integer, the higher the priority. Jobs are never interrupted, and each job is completed prior to starting another job and the current job is finished prior to the worker leaving for the day (although there may be unstarted jobs left in the queue). If the worker is currently working on a job and a higher priority job comes in, he/she won’t switch to the higher priority job until the current job is finished.

Whenever a job is started by the worker, a message is printed giving the job’s id, department and number of pages. Whenever a job is finished, its job id, department and number of pages is printed. Each time a job is put into a queue, a message is printed with this same information and also the priority number. When printing the queue, the priority, job id, department and pages is to be printed. Each different case is to be appropriately labeled so it can easily be determined what case the output relates to.

The worker in the copy center can be in one of two possible “states”: busy (making copies) or idle (nothing to do). He/she always start at the beginning of the day (start of the program) in the idle state. The program always starts with no jobs to do, no current job (left over jobs would be re-entered normally into the program after startup).

The program is an interactive command interpreter, which responds to the following single character commands:

- `r priority deptNum numCopies` – enters a new job into the system with the given priority. If the worker is idle, he/she will become busy and start the job, otherwise it will be placed onto the job queue.
- `f` – signifies worker finishes a job. If no more jobs available the worker becomes idle, otherwise worker will start oldest highest priority job available.
- `l` – lists the contents of the job queue, along with priority, in priority order.
- `d` – signifies worker is done for day. If busy, current job is finished. Either prints message indicating whether no more jobs are remaining, or prints list of remaining jobs if any.

Any other command is to be reported as a bad command and the rest of that command line is to be ignored. You may assume that if a command letter is correct, the form of the rest of the command will be correct.

There is a potential of a command error, even when only giving legal commands, depending on the state of the worker. For instance, the worker can’t finish a job if he/she is idle. The following table shows the action and/or error for all the possible combinations of state and command.
For the finish command, if no error, the resulting worker state depends on the state of the queue.

<table>
<thead>
<tr>
<th></th>
<th>non-empty</th>
<th>empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>starts job, busy</td>
<td>idle</td>
</tr>
</tbody>
</table>

What happens for a receive command depends on the state of the worker rather than the state of the queue (an idle worker will become busy), as seen in the first table. The list command does not do anything to worker state, and the done command terminates the program, so the resulting worker state is immaterial.

Although this seems complicated, the code is really relatively straightforward. You need a loop to input the commands – terminating when you get the done command. I recommend a procedure, called within the main input loop in the main program, to select the actual command (I suggest using a switch for selection). For each command, I recommend another procedure which actually processes the command. Although this results in a number of procedures, the actions for any one command are then localized, and the code is easier to understand and easier to debug. If you follow these recommendations, you will find that the main program (with the main input loop) and the procedure containing the switch are almost trivial. All the real work is done in the individual command procedures.

You are to use a programmer defined enumerated type to provide your two worker states.

```cpp
enum State {idle, busy};
```

This should be defined globally at the beginning of your “application” (non-class) code file (or placed in a header file and included if you split your application code across several code files). You will need a variable which contains the current worker state (it will be a parameter for most of your procedures).

Two header files have been provided in the course public area. You are implement these two classes and use them in your solution. The priority queue will hold copy job objects. You will also need a current job variable in the program (again, passed as a parameter to most procedures). Note: you can’t simply use the front element in the queue as the current job, as a new higher priority job might come in while the worker was still working on the original front element.

The priority queue is to be implemented as an ordered (by priority) chain of dynamically allocated elements. The priority is not part of the information stored, it is considered queue implementation information, and thus is stored separately from the copy job information. You are to implement the destructor, copy constructor, and assignment operator for the class. Note: the actual program code may not actually test the copy constructor or assignment operator. The copy constructor could be exercised if you passed the queue by value (which
would also test the destructor, as the copy would be destroyed when the function exits). Once my program was working, I temporarily used pass by value (where I had used pass by constant reference) to test these two operations, and then switched back to constant reference afterwards. It is doubtful you will use the assignment operator, but it does depend on copy constructor and destructor code, so, assuming correct high level assignment operator implementation and no typos, successfully exercising the other two would indicate the third would also be correct.

The copy job class needs a no argument constructor since you will be dynamically allocating elements for the chain (which will end up using no argument construction for the copy job portion of the element).

The only real “trick” in this assignment is how to get the unique ids for the copy jobs. The easiest way, assuming you follow the suggestion for a procedure to implement each command, is to have a static local integer variable defined in the procedure to handle the receive command. Initialize it to 0 (this initialization is done only once for the life of the program, even though the procedure may be invoked many times. Also define a new copy job variable, using the one argument constructor, incrementing the contents of the static variable in the process.

```c
static int id = 0;
CopyJob newJob(++id);
```

Then, enqueue this new job onto the queue or make it the new current job, depending on the worker’s state. This insures that each object is given a new id, and eliminates the need to pass an extra argument which provides the new id.

You will need to use “get” (in a loop) to input characters to flush the end of a bad command line. The standard input operator for characters won’t work, since it will skip over space characters, including the end of line character. “get”, on the other hand, inputs the next character regardless of whether it is a space character or not. This allows you to use the new line character at the end of the command line as a sentinel.

Note: you may create an extra Worker class, which encapsulates the current job and state into the worker object, if you desire. This may make your program simpler in some ways (fewer arguments, and the class code will handle some of logic), and more difficult in others (getting information cleanly into and out of the worker objects). If you take this approach, you still must implement and use the classes specified above.

Sample run

```
blj(mithrandir): a.out
enter commands, end with d

?> f
   not working on a job

?> r 8 111 23
   starting job 1 111 23

?> r 10 111 15
   putting job [2 111 15 ] onto queue with priority (10)

?> r 20 222 100
```
putting job [3 222 100 ] onto queue with priority (20)

?> 1
following jobs in queue
 20 -- 3 222 100
 10 -- 2 111 15

?> f
  finished job 1 111 23
  starting job 3 222 100

?> r 10 333 5
  putting job [4 333 5 ] onto queue with priority (10)

?> r 3 111 2
  putting job [5 111 2 ] onto queue with priority (3)

?> d
  finished job 3 222 100
  done for the day
remaining jobs
 10 -- 2 111 15
 10 -- 4 333 5
 3 -- 5 111 2

blj(mithrandir):