**Programming Assignment 4: Stacks & Queues**

*Due: 11:59 pm the day before your lab meets the week of October10th.*

## 1. Introduction

The purpose of this lab is to simulate an office by managing the document flow using stacks and queues. This program is a good example of how many programs exist to model activities in the real world. The situation you will write a program to model is that of a single *office* with a single *worker*, and is illustrated in Figure 1. The *office* has an *in-box* where *documents* are *submitted*, and which is modeled by a stack because each *document* submitted is put in the *in-box* on top of the others inside it. The worker *enters* and *leaves* the *office* at will. When in the *office*, the *worker* may be *idle*, *reading* a *document*, or *sorting* the *in-box*. Each *document* has a *name* and a *priority* of : 1, 2, or 3, with 1 being best. When the *in-box* is *sorted*, each *document* is placed in a *queue* corresponding to its *priority*. When the *worker* wishes to begin *reading*, they *choose* what to read by *removing* the document at the front of the highest priority, non-empty, *queue*. The worker never *leaves* the office or *sorts* the *in-box* before they finish *reading* the current *document*. Occasionally, those who submit a document *withdraw* it, which means it must be *found* in the *in-box* or one of the priority *queues* and *removed*. We will assume for simplicity that a document will not be *withdrawn* while the worker is *reading* it. However, your program should be robust enough to handle a *withdrawal* request for a *document* that is no longer in the *in-box* or priority *queues*.

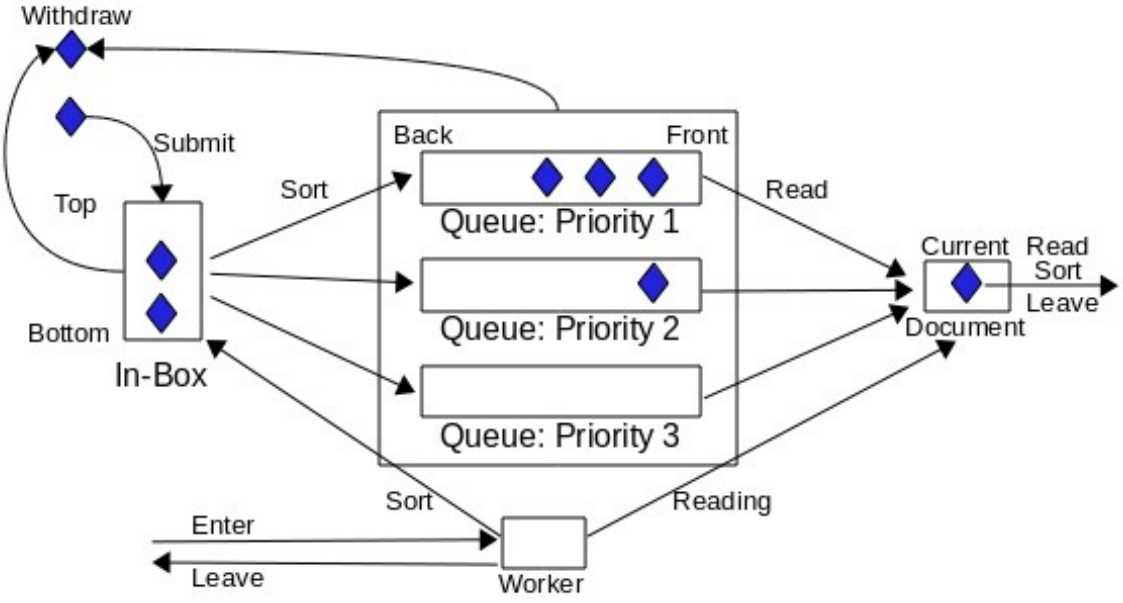


Figure 1: Basic Workflow and Commands in Office Simulation

You will complete the starter code provided to implement a simulation of the office according to the semantics of the office described above. The elements of the office and the various operations your program will model are put in italics in the paragraph describing the office as an aid to your making an inventory of the elements, actions, and states of the worker that your program should represent. While the description is intended to be complete, you should think about the situation on your own and form your own opinions about how things work, thus filling in any gaps or inaccuracies.

At the top level, there will be an ADT to represent the *Office* as a whole, which will contain an in-box stack and three document queues. The Office ADT will also track whether the *worker is in* the office or *not*, what the *current* document being read is, if any, and the *number* of documents *read* by the worker during each period spent in the office. The ADTs *Stack* and *Queue* will manage sets of documents according to their specified semantics. The ADT *Document* will contain document information, such as name and priority.

Your program will process commands to *submit* and *withdraw* documents to the office. Submission places the document on the top of the in-box, but withdrawal can be made from either the in-box or the priority queues. The sequence of commands represent the sequence of events in the office, and are discussed in greater detail later in this document.

## 2. Getting Started

Your program will read input from the file whose name is specified by the first argument on the command line. The main.cpp program provided illustrates how to process the input commands. It also provides a way to help ensure that output from programs by all students can be compared character by character with the reference output by the make file using the "diff" command. Look at the "test1" target in the make file.

The API for the office is also provided, since it tells you little beyond what the set of commands defined below tell you. Further, the "<<" operator routines for the Stack and Queue ADTs, and the **print( )** method for the List ADT are also provided to help ensure that the output of your program will match that of the reference file when your program is properly implemented.

The following is an example of how your program will be run from the command line:

```
$ ./office input.txt
```

Your program will include at least five (5) classes: Office, Stack, Queue, LinkedList, and Document.

### 2.1. Office

This ADT represents the state of the office as a whole. It should track the presence and activity of the worker and the number of documents read during each period the worker spends in the office.

Most, but not all, of the methods for the class correspond to the set of actions taken in response to a command in the input file. Methods for this class should include:

**void enter( )**
This routine is executed when the worker enters the office. If the office state is printed when the worker is not there, the READER state should be given as ABSENT.

**int get_read_count( )**
The duty of this routine is painfully obvious. The read_count variable should count the number of documents read by the worker while one period of being in the office.

**void leave( )**
The worker leaves the office. This signals that the worker is finished reading the current document, if any.

**void read_document( )**
This routine represents the worker starting to read a new document. The next document should be properly selected, and what document is currently being read should be tracked.

**void sort_inbox( )**
This routine is executed when the next command indicates it is time to sort the in-box. Items are taken from the stack according to its specified semantics, and placed in the priority queues according to theirs.

**void submit_document(std::string name, std::string priority)**
This routine is executed when a new document is submitted to the office. It should be placed in the in-box according to its specified semantics.

**void withdraw_document(std::string name)**
This routine is called when the next command withdraws the named document from the office. It must find the document in the in-box, or one of the priority queues, and remove it.

**friend std::ostream& operator<<(std::ostream &os, Office &office)**
This code is provided in what we believe is its complete form, which should ensure standard output from every student's program. If you come to believe you must change this routine, talk to the GTA or instructor first.

### 2.2. Stack

This ADT implements the basic semantics of a stack, and is used to simulate the in-box in the office. In addition, it needs to be able to support the withdraw operation, which goes beyond the standard stack semantics.

Methods for this class should include:

**void push(Document* document)**
Place a document in the in-box on the top of the pile of documents already there, if any.

**Document *pop()**
Take the top document from the pile in the in-box.

**bool is_empty()**
Return True if the in-box is empty, False otherwise.

**bool withdraw(std::string name)**
Withdraw the named document from the in-box, if present.

**friend std::ostream& operator<<(std::ostream &os, Stack &in_stack);**
This routine is a very thin wrapper for the print( ) routine of the Linked List ADT used to implement the stack.

### 2.3. Queue

This ADT should implement the basic queue semantics, plus the ability to withdraw a named document. Three instances of this ADT should be used for the three document priority levels.

Methods for this class should include:

**Document* dequeue( )**
Remove the document from the front of the queue.

**void enqueue(Document* document)**
Add a the document to the end of the queue.

**bool is_empty( )**
Return True if the queue is empty, and False otherwise.

**bool withdraw(std::string name)**
Look in the queue for a document with the name indicated and remove it if found.

**std::ostream& operator<<(std::ostream &os, Queue &in_queue)**
This routine is a very thin wrapper for the print( ) routine of the Linked List ADT used to implement the queue.

### 2.4. LinkedList

This is the core of the code in many ways. Since this ADT is being used to support both the Stack and Queue ADTs, adn that we need to be able to withdraw a document from any point on the list, we assume the index-based API used in the textbook which makes all of those relatively convenient. However, if you prefer to use a slightly different API to support the stack, that should not affect the output of the program and our ability to “diff” the output of your program against the reference output. Showing a little creativity in this section of the project would be educational.

Methods for this class in our reference solution include:

**int get_size()**
This routine returns the number of elements currently in the list. This is helpful in using the indexed-based API for this solution. A different solution, one with both a head and tail pointer for each queue, for example, might not need this method.

**void insert(int index, Document* document)**
This routine inserts at a given position in the list. In an implementation with both a head and tail pointer, the interface for this routine might change to use a parameter to indicate whether the insertion should be at the beginning or the end of the list.

**bool is_empty()**

Return True if the list is empty, False, otherwise.

**void print(std::ostream& os)**

This routine steps down the list and prints the name of each item with a space between them. It is provided to to to help ensure the “diff” of the output of your program against the reference output is more likely to be valid.

**void remove(int index)**

Remove an element of the list at a specific position. Another approach to this assignment might use a head and tail pointer and use the parameter to indicate whether to remove from the head or the tail of the list.

**bool remove(std::string name)**

Remove the item on the list with the specified name, if it is present. Note that this and the other remove( ) routine are an example of method overloading, and that the compiler distinguishes them from each other using the type and order of their arguments.

**Document *retrieve(int index)**

Return the pointer to the document at the specified index in the list. If you were using a head and tail pointer implementation of the list then retrieving wither the head or the tail might be sufficient.

### 2.5. Document

The document class represents each document moving through the office. Each has a name and a priority.

Methods for this class should include:

**Document(std::string in_name, std::string in_priority)**

The constructor initializes the name and priority fields of the instance of the class. You could make other approaches work, but this was the easiest in the context of out example solution.

**std::string get_name( )**

What this routine does is too obvious to explain. If you need to, ask a question about this of the instructor or the GTA.

**Priority get_priority( )**

Again, too obvious to explain.

**std::ostream& operator<<(std::ostream &os, Document &document)**

This routine is provided to help ensure consistent output from all implementations. There is one interesting point here. Note that the priorities for documents are defined as an enumeration. Enumerations are a way to give symbolic names to a set of choices without having to worry about the underlying values themselves. Enumerations are technically a separate type, but use integer values internally, and for printing purposes.

The interface uses values 1, 2,and 3. This routine adds 1 tot he value, as enumerations assign values starting at zero. This routine is not currently used in the reference solution, but was used during debugging and we provide it for your henefit if you find it useful during development.o

**Priority convert_priority(std::string str_priority)**

This routine converts a priority value given as a string on a 1, 2, 3 scale to the enumeration values defined.

### 3. Input & Output

Your program will be expected to handle the following commands:

**enter**

The worker enters the office.

**leave**

The worker leaves the office. This also marks the end of reading the current document, if the worker is currently reading when this command is encountered.

**print**

Print the current state of the office.

**read**

Read the next document as determined by the state of the various document queues. Select the front of the highest priority non-empty queue.

**submit** *document-name document-priority*

Add a document with the given name and priority to the in-box.

**sort-inbox**

The worker finishes reading the current document, if there is one, and then takes the documents out of the in-box and adds them to the relevant priority queue until the in-box is empty.

**withdraw** *document-name*

Take the relevant document out of the in-box, or the priority queues, if it is present.

#### 3.1. Sample Input

The input given below is the reference input used by “bash> make test1” to drive your program, which produces output it compares to the reference output for correctness. Using this input, you should be able to step through the logic of the simulation, know the state of the in-box, the queues, and the reader after each command is executed. This will also let you predict the output of the program, and is exactly the understanding you need to be able to implement the logic of the program to simulate the specified office behavior.

```
enter
submit A 1
submit B 2
submit C 3
submit D 1
submit E 1
submit F 3
submit G 2
submit H 1
print
withdraw D
leave
enter
sort-inbox
print
withdraw A
print
read
print
read
submit I 1
print
read
submit J 1
read
print
sort-inbox
print
submit K 3
submit L 2
read
submit M 2
leave
submit N 2
submit O 2
submit P 2
print
```

### 4.2. Sample Output

This is the output produced by the reference solution. The code provided includes all the print statements used by the solution to produce the output. If the semantics of your simulation matches that of the reference solution, then the output should match. Some differences may be trivial, an extra blank line, or a slight format change. If the sequence of information in the output of your program is different than that of the reference output then there is a difference in logic between your implementation and the reference solution.

```
ENTERED
Submit: A Priority: 1
Submit: B Priority: 2
Submit: C Priority: 3
Submit: D Priority: 1
Submit: E Priority: 1
Submit: F Priority: 3
Submit: G Priority: 2
Submit: H Priority: 1
```

```
-------------------------
| CURRENT OFFICE STATUS |
-------------------------
READER : IDLE
INBOX  : A B C D E F G H <-TOP
QUEUE 1: EMPTY
QUEUE 2: EMPTY
QUEUE 2: EMPTY

Document D removed from inbox
LEFT
-------------------------
| CURRENT OFFICE STATUS |
-------------------------
READER : IDLE
INBOX  : A B C E F G H <-TOP
QUEUE 1: EMPTY
QUEUE 2: EMPTY
QUEUE 2: EMPTY
READCNT: 0

ENTERED
In Box Sorted
-------------------------
| CURRENT OFFICE STATUS |
-------------------------
READER : IDLE
INBOX  : EMPTY
QUEUE 1: A E H <-FRONT
QUEUE 2: B G <- FRONT
QUEUE 3: C F <- FRONT

Document A removed from queue 1
-------------------------
| CURRENT OFFICE STATUS |
-------------------------
READER : IDLE
INBOX  : EMPTY
QUEUE 1: E H <-FRONT
QUEUE 2: B G <- FRONT
QUEUE 3: C F <- FRONT

READER : START H

-------------------------
| CURRENT OFFICE STATUS |
-------------------------
READER : READING H
INBOX  : EMPTY
QUEUE 1: E <-FRONT
QUEUE 2: B G <- FRONT
QUEUE 3: C F <- FRONT

READER : START E

Submit: I Priority: 1
```

```
-------------------------
| CURRENT OFFICE STATUS |
-------------------------
READER : READING E
INBOX  : I <-TOP
QUEUE 1: EMPTY
QUEUE 2: B G <- FRONT
QUEUE 3: C F <- FRONT

READER : START G

Submit: J Priority: 1
READER : START B

-------------------------
| CURRENT OFFICE STATUS |
-------------------------
READER : READING B
INBOX  : I J <-TOP
QUEUE 1: EMPTY
QUEUE 2: EMPTY
QUEUE 3: C F <- FRONT

In Box Sorted
-------------------------
| CURRENT OFFICE STATUS |
-------------------------
READER : IDLE
INBOX  : EMPTY
QUEUE 1: I J <-FRONT
QUEUE 2: EMPTY
QUEUE 3: C F <- FRONT

Submit: K Priority: 3
Submit: L Priority: 2
READER : START J

Submit: M Priority: 2
LEFT
-------------------------
| CURRENT OFFICE STATUS |
-------------------------
READER : IDLE
INBOX  : K L M <-TOP
QUEUE 1: I <-FRONT
QUEUE 2: EMPTY
QUEUE 3: C F <- FRONT
READCNT: 7

Submit: N Priority: 2
Submit: O Priority: 2
Submit: P Priority: 2
-------------------------
| CURRENT OFFICE STATUS |
-------------------------
READER : IDLE
INBOX  : K L M N O P <-TOP
QUEUE 1: I <-FRONT
QUEUE 2: EMPTY
```

```
QUEUE 3: C F <- FRONT
```

## 5. Grading criteria

| | | |
|---|---|---|
| 40% | 24 pts | Correctness of Linked List, Stack, and Queue Class implementations |
| | Stacks | 7 pts |
| | Queues | 7 pts |
| | Linked Lists | 10 pts |
| 30% | 18 pts | Correctness of Command logic in Office methods |
| 15% | 9 pts | Programming style and output format match |
| 15% | 9 pts | Documentation |
| 100% | 60 pts | Total |