Fundamental Computer Programming - C++ Lab(II)

Lab 4: Queue Class –Task Scheduling

03/10/2022

Rung-Bin Lin

International Bachelor Program in Informatics
Yuan Ze University

Purposes of this Lab

- **Constructors and destructors.**
- **Default memberwise assignment.**
- **More practices on class**

Destructor

- A destructor is a special member function. The name of the destructor for a class is the name of the class prefixed with a tilde character ~. For example, **~Time()** is the destructor of Time class and **~Stack()** is the destructor of Stack class.
- A class's destructor is called implicitly when an object is destroyed. This occurs, for example, as an automatic object is destroyed when program execution leaves the scope in which that object is created.
- When an object is destroyed, the memory allocated to the object is not released. It can still be reused to hold new objects.
- A class may have only one destructor-destructor overloading is not allowed.
- A destructor must be public.

More on Destructor

- Constructors and destructors are called implicitly by the compiler.
- Generally, destructor calls are made in the reversed order of the corresponding constructor calls.
- Constructors are called for objects defined in the global scope before any other function in that file begins execution.
- Refer to 9.10 for more about destructors

Default Memberwise Assignment

- The assignment operator (=) can be used to assign an object to another object of the same type. By default, such assignment is performed by **memberwise** assignment. That is, each data member of the object on the right of the assignment operator is assigned individually to the same data member in the object on the left of the assignment operator. For example with Time class

```
Time t1;
Time t2;
t1 = t2;
```

Is equivalent to

```
Time t1;
Time t2;
t1.hour = t2.hour;
t1.minute = t2.minute;
T1.second = t2.second;
```

Lab 4: Task Scheduling with Queues

- This project has two parts. The first part creates a Queue class with the following requirements
  - **private data members:**

    int size; // the size of a queue
    int *intQueue; // the pointer to a dynamically created array
    int front; // index to the front most element in a queue
    int rear; // index to the rear most element in a queue
    int count; // the number of elements in a queue;
  - **public member functions:**

    Queue(int = 10); // constructor with queue size initialized to 10
    ~Queue(); // destructor
    void enqueue(int); // Insert an element in the rear end of a queue
    int   dequeue(); // remove and then return the front most element in the queue
    int peek(); // return the front most element in the queue
    int getCount(); // return the number of elements stored in the queue
    int getSize(); // return queue size
    bool isEmpty(); // return true if queue is empty
    bool isFull(); // return true if queue is full
    void clearQueue; // clear the content of queue and reset front, rear, and count
    void printQueue(); // print the content of a queue from front to rear.

Problem Description cont.

- The second part should use the Queue class to solve the following problem.
  - Suppose a CPU has N tasks numbered from 1 to N to be completed one by one according to an ideal order. Assume some of the tasks are put into a waiting list in the beginning according to their arriving order. When a task gets to the front end of the waiting list and is the first yet-to-be executed task in the ideal-order list, it is executed. Otherwise, it must be rescheduled to the end of the waiting list. If a task appears in the ideal-order list but not in the waiting list, it should be treated as if it has been executed. Assume that executing a task takes 3 seconds and rescheduling a task takes 2 seconds. You have to write a program to compute the total time taken by the CPU to complete all the tasks. Note that there may have some tasks which are not in the waiting list in the beginning.

Example

- Assume there are four tasks numbered from 1 through 4. Given their ideal order 3, 1, 4, 2 and their arriving order 2, 1, 3, the tasks will be executed as follows:
  - Since the first task in the waiting list, i.e., task 2 is not the first yet-to-be executed tasks in the ideal-order list, it must be rescheduled. This takes 2 seconds. Now, the waiting list is updated into 1, 3, 2.
  - Since the first task in the waiting list, i.e., task 1 is not the first yet-to-be executed task, it must be rescheduled. This takes 2 seconds. Now, the waiting list is updated into 3, 2, 1.
  - Since the first task in the waiting list, i.e., task 3 is the first yet-to-be executed task, it is executed. This takes 3 seconds. Now, the waiting list is updated into 2, 1.
  - Since the first task in the waiting list, i.e., task 2 is not the first yet-to-be executed task, it is rescheduled. This takes 2 seconds. Now, the waiting list is updated into 1, 2.
  - Since the first task in the waiting list, i.e., task 1 is the first yet-to-be executed tasks, it is executed. This takes 3 seconds. Now, the waiting list is updated into 2.
  - Since the first task in the waiting list, i.e., task 2 is the first yet-to-be executed task, it is executed. This takes 3 seconds. Now, the waiting list is empty. Totally the CPU takes 15 seconds to complete the three tasks.

- Other requirements
  - Separate the class definition, class implementation, and main function into different files Queue.h, Queue.cpp, and main.cpp (for main function) as shown in Figs. 9.3, 9.4, and 9.5, respectively.
  - The main() should not contain the cout statement that prints *-- Warning: Queue is EMPTY now, -- Warning: Queue is FULL now, -- A queue of size … is created, -- A queue of size … is destroyed.*

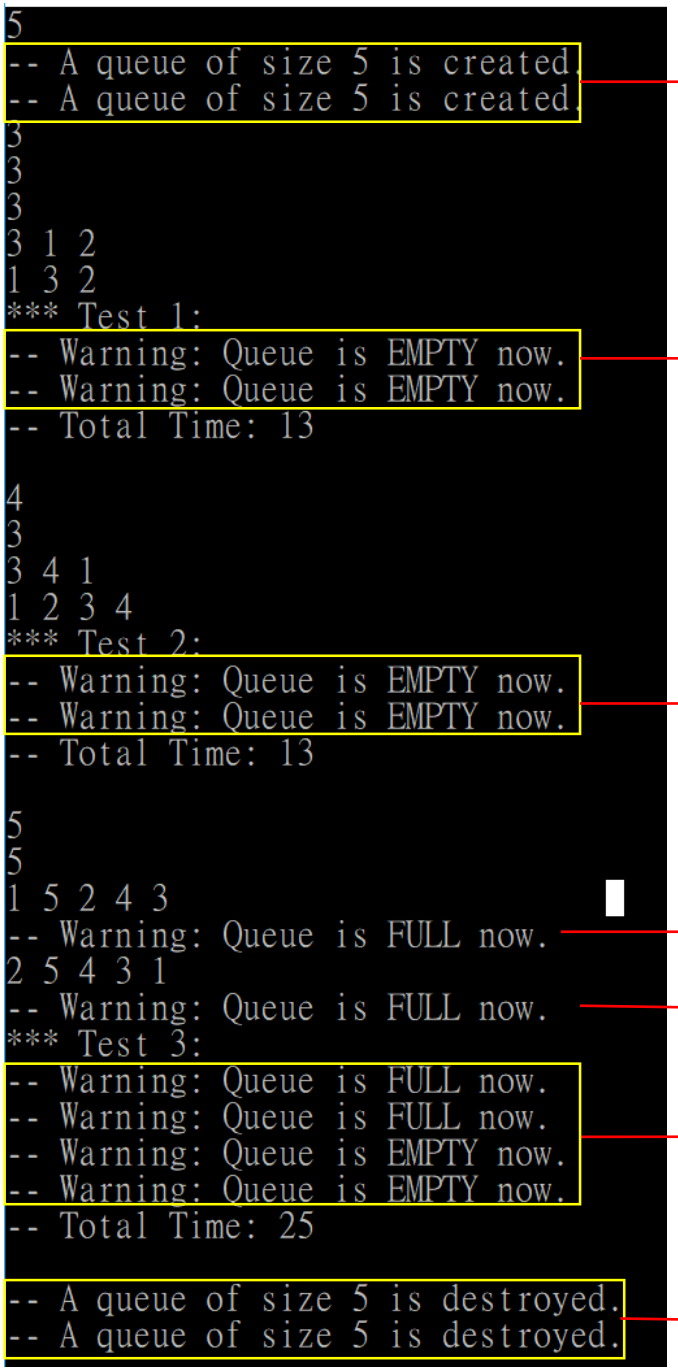
Input & Output Formats

- **Input format**
  - The first line gives the size of the queue. The second line gives the number of test cases. Each test case takes four lines. The first line of a test case gives a number 1<=N<=100, denoting the number of tasks. The second line of a test case gives the actual number of tasks that are in the arriving-order list. The third line specifies the arriving order. The fourth line of a test case gives the ideal order of the tasks. All the tasks will be in the ideal-order list of the input. Note that the size of the queue should be greater than the number of tasks in each test case.
- **Output format**
  - Refer to the output examples to see what is printed for each test case.

Hints

- You can create two queues. One is to store the ideal order. The other is to store the waiting list. Rescheduling the waiting list by first doing dequeue() and then doing enqueue(). If the task numbers in the front end of these two queues are the same, added up the total time and remove this task from these two queues. Certainly, for this you have to consider the tasks which do not appear in the waiting list. Continue rescheduling the waiting list until the waiting-list queues is empty.
- You can use either an array or a linked list to implement a queue. If a linked list is used, the int type in this line int *intQueue; can be changed into the type used to create a node in the linked list.

Input & Output Example



These two lines should be in the output.

These two lines should be in the output.

The Total Time printed must be correct.

At least, one of these two lines should be in the output.

This line should be in the output.

This line should be in the output.

These four lines should be in the output.

These two lines should be in the output.

Key Points for Grading

- The output should be exactly the same as given in the output example.
- The class declaration should be exactly the same as that specified for this lab.
- All the member functions should be implemented.
- The output should be correct.
- The output messages about creation and destroy of queues should be printed also.
- Class definition, class implementation, and main function should be stored into different files Queue.h, Queue.cpp, and main.cpp, respectively.