



# **Fundamental Computer Programming - C++ Lab(II)**

---

## **Lab 4: Queue Class –Task Scheduling**

03/18/2021

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

- In this project you have to create 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 the 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.

## • 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.

## Problem to be solves using a queue from Queue class

- Suppose a CPU has N tasks numbered from 1 to N to be completed one by one according to an ideal order. Assume all the tasks are put into a waiting list in the beginning according to their arriving order. When a task gets to the front of the waiting list and is the one in the front of the **yet-to-be executed tasks** in the ideal order, it is executed. Otherwise, it must be rescheduled to the end of the waiting list. Suppose the execution time of a task is 2 seconds and the time of a task being rescheduled takes 1 second. You have to write a program to compute the total time taken by the CPU to complete all the tasks. Your program must use the Queue class created above to solve the problem.
- Example:
  - Given three tasks whose ideal order is 3, 1, 2 and whose arriving order is 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 one in the front of the yet-to-be executed tasks in the ideal order, it must be rescheduled. This takes 1 second. 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 one in the front of the yet-to-be executed tasks in the ideal order, it must be rescheduled. This takes 1 second. Now, the waiting list is updated into 3, 2, 1.
    - Since the first task in the waiting list, i.e., task 3 is the one in the front of the yet-to-be executed tasks in the ideal order, it is executed. This takes 2 second. Now, the waiting list is updated into 2, 1.
    - Since the first task in the waiting list, i.e., task 2 is not the one in the front of the yet-to-be executed tasks in the ideal order, it is rescheduled. This takes 1 second. Now, the waiting list is updated into 1, 2.
    - Since the first task in the waiting list, i.e., task 1 is the one in the front of the yet-to-be executed tasks in the ideal order, it is executed. This takes 2 second. Now, the waiting list is updated into 2.
    - Since the first task in the waiting list, i.e., task 2 is the one in the front of the yet-to-be executed tasks in the ideal order, it is executed. This takes 2 second. Now, the waiting list becomes empty. Totally the CPU takes 9 seconds to complete the three tasks.
  - Input format:
    - The first line gives the size of the queue. The second line gives the number of test cases. Each test case takes three lines. The first line of a test case gives a number  $1 \leq N \leq 100$ , denoting the number of tasks. The second line of a test case gives the **ideal order** of the tasks. The third line of a test case specifies the **arriving order** of the tasks. Note that the size of the queue should be greater than the number of tasks in that test case.
  - Output format:
    - For each test case, first print test number on one line, then print the waiting-list content on a line **after** a task is rescheduled, and lastly print the total time taken by the CPU to complete all the tasks on one line. In addition, [When a queue is created or destroyed, information as shown in the input & output example](#)  should be printed.



# 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 of these two queues are the same, added up the total time and remove this task from these two queues. Continue rescheduling the waiting list until the two queues are empty.

# Input & Output Example

Input	Output
5	A queue of size 5 is created.
3	A queue of size 5 is created.
3	Test 1:
3 1 2	3 2 1
1 3 2	1 2
4	8
3 2 4 1	Test 2:
1 2 3 4	2 3 4 1
5	3 4 1 2
1 5 4 3 2	1 2 4
2 5 4 3 1	2 4 1
	12
	Test 3:
	5 4 3 1 2
	4 3 1 2 5
	3 1 2 5 4
	1 2 5 4 3
	5 4 3 2
	15
	A queue of size 5 is destroyed.
	A queue of size 5 is destroyed.

# Key Points for Grading

- 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 information about creation and destroy of queues should be printed also.