

Test 3 Review

CSC 112 Fall 2009

Objects

Here's a header file for a Queue class I am writing.

```
class Queue
{
    public:
        Queue(int capacity);
        Queue(const Queue& copyQueue);
        ~Queue();
        void add(const double& valueToAdd);
        double* delete(const double& valueToDelete);
    private:
        double* arrayOfData;
        int front, rear;
        int size;
}
```

Objects

You are interested in finishing my Queue object and are happy to see it is built on top of arrays, as you seriously dislike LinkedList programming.

The Queue object should have a fixed capacity (max number of items that can be held), but the user should be able to define that capacity at runtime when the Queue object is created. An empty array based queue will have a front index of -1 and a rear index of -1. The front index is always one index in front of the first item in the queue and the rear index always sits on the last item in the queue.

Write the appropriate single-parameter (int capacity) Queue constructor and the destructor given these requirements.

Objects - Answers

```
Queue::Queue(int capacity)
```

```
{
```

```
}
```

```
Queue::~~Queue()
```

```
{
```

```
}
```

Objects - Answers

```
Queue::Queue(int capacity)
```

```
{
```

```
    size = capacity; // this interpretation sets size to be = capacity  
    // an alternative interpretation could use size as # of items actually  
    // in the list (rear – front)
```

```
    arrayOfData = new double[size];
```

```
    front = rear = -1;
```

```
}
```

```
Queue::~~Queue()
```

```
{
```

```
    delete [] arrayOfData;
```

```
    front = rear = -1; // optional, but good reminder for =
```

```
    size = 0; // optional, but good reminder for =
```

```
}
```

Objects

Now define the copy constructor and assignment operator. These should perform their usual functions, copying the data from an input Queue into the current Queue.

```
// copy constructor
Queue::Queue(const Queue &copyQueue)
{

}
}
```

Objects

Now define the copy constructor and assignment operator. These should perform their usual functions, copying the data from an input Queue into the current Queue.

```
// copy constructor
Queue::Queue(const Queue &copyQueue)
{
    size = copyQueue.size;
    arrayOfData = new double[size];
    front = copyQueue.front;
    rear = copyQueue.rear;
    // this attempts to copy over just the meaningful data (between front
    and rear) ... arguably, you could just copy over the whole array
    for (int j = front+1; j <= rear; j++)
        { arrayOfData[j] = copyQueue.arrayOfData[j]; }
}
```

Objects

Now define the copy constructor and assignment operator. These should perform their usual functions, copying the data from an input Queue into the current Queue.

```
// assignment operator
Queue& Queue::operator=(const Queue &copyQueue)
{

}
}
```

Objects

Now define the copy constructor and assignment operator. These should perform their usual functions, copying the data from an input Queue into the current Queue.

```
// assignment operator
Queue& Queue::operator=(const Queue &copyQueue)
{
    if (this != &copyQueue) {
        delete [] arrayOfData; // clean up

        size = copyQueue.size;
        arrayOfData = new double[size];
        front = copyQueue.front;
        rear = copyQueue.rear;
        // again, just copying over "valid" data, but could really copy over whole
        // array and it would be OK
        for (int j = front+1; j <= rear; j++)
            { arrayOfData[j] = copyQueue.arrayOfData[j]; }
    }
    return *this;
}
```

Objects

Assume you want to overload the + operator so that it also performs a Queue add. What would the new function signatures need to be so that

myQueue + 2.0

2.0 + myQueue

would both insert 2.0 at the rear of the queue.

Objects

Assume you want to overload the + operator so that it also performs a Queue add. What would the new function signatures need to be so that

myQueue + 2.0
2.0 + myQueue

would both insert 2.0 at the rear of the queue.

To get “2.0 + myQueue”:

```
friend Queue operator+(const double& value, const  
Queue & queue);
```

and to get “myQueue + 2.0”:

```
Queue Queue::operator+(const double& value);
```

Objects

I want to argue that almost any member operator function could be rewritten as:

```
friend returnType operator???(const  
    ClassOfInterest & x, const SomeOtherType &  
    y)
```

Will you support me in my belief? Why or why not?

Objects

I want to argue that almost any member operator function could be rewritten as:

```
friend returnType operator???(const  
    ClassOfInterest & x, const SomeOtherType &  
    y)
```

Will you support me in my belief? Why or why not?

You should believe me.. This is essentially equivalent to the member function approach (supporting a variable of our own type on the left-hand-side of the + operator)

Linked Lists

By reading the code below, indicate the purpose of the following member function for standard head pointer linked lists.

```
Node* LinkedList::doSomething()
{
    if (head == 0) return 0;
    else
    {
        Node* current = head;
        Node* temp1;
        Node* temp2;
        while (current != 0)
        {
            if (head == current)
            { temp2 = new Node(current); temp2->next = 0;}
            else
            {
                temp1 = temp2;
                temp2 = new Node(current);
                temp2->next = temp1;
            }
            current = current-> next;
        }
        return temp2;
    }
}
```

Linked Lists

By reading the code below, indicate the purpose of the following member function for standard head pointer linked lists. **THIS GIVES YOU A POINTER TO A NEW LIST THAT IS THE REVERSE OF THE ORIGINAL.**

```
Node* LinkedList::doSomething()
{
    if (head == 0) return 0;
    else {
        Node* current = head;
        Node* temp1;
        Node* temp2;
        while (current != 0) //looping down the current list
        {
            if (head == current)
                { temp2 = new Node(current); temp2->next = 0;} // copy head of orig, it's now
                back of new list because it points to 0.
            else { // as we move down orig list, the new items are being put on front of new
                list, basically making a reversed list
                    temp1 = temp2; // as
                    temp2 = new Node(current);
                    temp2->next = temp1;
                }
            current = current-> next;
        }
        return temp2;
    }
}
```

Linked Lists

- Suggest how an `insertInSortedOrder` method for `LinkedLists` is a natural analogue for insertion sort on arrays.

Linked Lists

- Suggest how an `insertInSortedOrder` method for `LinkedLists` is a natural analogue for insertion sort on arrays.

In insertion sort for arrays, we are moving a new unsorted item through a group of sorted data. We bypass anything we are smaller than but stop when we hit something smaller than us.

For `insertInSortedOrder`, one would need to assume the linked list was already sorted and we are adding one new element. Basically, we will slide past all of the elements we are smaller/bigger than (depending on direction) but stop once we hit an entry we can't pass or one of the ends of the list - so basically it is the same process as `insertionSort` for arrays. `insertInSortedOrder` is actually even a little nicer, as we don't need to do any shifting as we do in arrays. Once we find the right spot in the linked list, we just open up a spot by breaking the link that is there and restructuring the link.

this

Is the following a valid use
of the *this* keyword?

Why or why not?

```
AClass::AClass(int aVariable)
{
    this.aVariable = aVariable;
}
```

```
class AClass
{
    public:
        AClass(int aVariable);
    private:
        int aVariable;
};
```

this

Is the following a valid use
of the *this* keyword?
Why or why not?

```
AClass::AClass(int aVariable)
{
    this.aVariable = aVariable;
}
```

```
class AClass
{
    public:
        AClass(int aVariable);
    private:
        int aVariable;
};
```

This is not valid...

The attempted concept is OK, as it allows you to reference the `aVariable` which is part of the class instead of the parameter called `aVariable`

However, ***this*** is a pointer, So you would need to use `this->aVariable` instead of `this.aVariable`; alternatively, use `(*this).aVariable`