Activity No. 5.1	
QUEUES	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:**
Section: CPE21S4	**Date Submitted:**
Name(s): CALVIN EARL PLANTA	**Instructor: Prof. Sayo**

6. Output

Using STL

```
C/C++
#include <iostream>
#include <queue>
using namespace std;

class Queue {
private:
	queue<int> q;

public:
	void enqueue(int element) {
	q.push(element);
	cout << element << " added to the queue." << endl;
	}

	void dequeue() {
	if (q.empty()) {
		cout << "Queue is empty! Cannot dequeue." << endl;
		return;
	}
	cout << q.front() << " removed from the queue." << endl;
	q.pop();
	}

	int peek() const {
	if (q.empty()) {
		cout << "Queue is empty!" << endl;
		return -1;
	}
	return q.front();
	}

	bool isEmpty() const {
	return q.empty();
	}

	int getSize() const {
	return q.size();
	}

	void display() const {
```

```
        if (q.empty()) {
                cout << "Queue is empty!" << endl;
                return;
        }

        cout << "Queue elements: ";
        queue<int> tempQueue = q;
        while (!tempQueue.empty()) {
                cout << tempQueue.front() << " ";
                tempQueue.pop();
        }
        cout << endl;
        }
};

int main() {
        Queue q;

        q.enqueue(10);
        q.enqueue(20);
        q.enqueue(30);
        q.enqueue(40);
        q.enqueue(50);

        cout << "Front element: " << q.peek() << endl;
        cout << "Current size: " << q.getSize() << endl;

        q.display();

        q.dequeue();
        q.dequeue();

        cout << "Front element after two dequeues: " << q.peek() << endl;
        cout << "Current size after dequeues: " << q.getSize() << endl;

        q.display();

        q.enqueue(60);
        cout << "Front element after adding 60: " << q.peek() << endl;

        q.display();

        return 0;
}
```

Output

```
/tmp/DTUcFOJbEy.o
10 added to the queue.
20 added to the queue.
30 added to the queue.
40 added to the queue.
50 added to the queue.
Front element: 10
Current size: 5
Queue elements: 10 20 30 40 50
10 removed from the queue.
20 removed from the queue.
Front element after two dequeues: 30
Current size after dequeues: 3
Queue elements: 30 40 50
60 added to the queue.
Front element after adding 60: 30
Queue elements: 30 40 50 60


=== Code Execution Successful ===
```

Table 5-1. Queues using C++ STL

Using Linked List

C/C++

```
#include <iostream>
using namespace std;

class Node {
public:
	int data;
	Node* next;

	Node(int value) : data(value), next(nullptr) {}
};

class Queue {
private:
	Node* front;
	Node* rear;
	int size;
```

```
public:
	Queue() {
	front = nullptr;
	rear = nullptr;
	size = 0;
	}

	~Queue() {
	while (!isEmpty()) {
		dequeue();
	}
	}

	void enqueue(int element) {
	Node* newNode = new Node(element);
	if (isEmpty()) {
		front = rear = newNode;
	} else {
		rear->next = newNode;
		rear = newNode;
	}
	size++;
	cout << element << " added to the queue." << endl;
	}

	void dequeue() {
	if (isEmpty()) {
		cout << "Queue is empty! Cannot dequeue." << endl;
		return;
	}
	Node* temp = front;
	cout << front->data << " removed from the queue." << endl;
	front = front->next;
	delete temp;
	size--;
	if (front == nullptr) {
		rear = nullptr;
	}
	}

	int peek() const {
	if (isEmpty()) {
		cout << "Queue is empty!" << endl;
		return -1;
	}
	return front->data;
	}

	bool isEmpty() const {
	return size == 0;
	}

	int getSize() const {
	return size;
	}
```

```
        void display() const {
        if (isEmpty()) {
                cout << "Queue is empty!" << endl;
                return;
        }
        Node* temp = front;
        cout << "Queue elements: ";
        while (temp != nullptr) {
                cout << temp->data << " ";
                temp = temp->next;
        }
        cout << endl;
        }
};

int main() {
        Queue q;

        q.enqueue(10);
        q.enqueue(20);
        q.enqueue(30);
        q.enqueue(40);
        q.enqueue(50);

        cout << "Front element: " << q.peek() << endl;
        cout << "Current size: " << q.getSize() << endl;

        q.display();

        q.dequeue();
        q.dequeue();

        cout << "Front element after two dequeues: " << q.peek() << endl;
        cout << "Current size after dequeues: " << q.getSize() << endl;

        q.display();

        q.enqueue(60);
        cout << "Front element after adding 60: " << q.peek() << endl;

        q.display();

        return 0;
}
```

Output

```
/tmp/I7pef0Hbt0.o
10 added to the queue.
20 added to the queue.
30 added to the queue.
40 added to the queue.
50 added to the queue.
Front element: 10
Current size: 5
Queue elements: 10 20 30 40 50
10 removed from the queue.
20 removed from the queue.
Front element after two dequeues: 30
Current size after dequeues: 3
Queue elements: 30 40 50
60 added to the queue.
Front element after adding 60: 30
Queue elements: 30 40 50 60
30 removed from the queue.
40 removed from the queue.
50 removed from the queue.
60 removed from the queue.


=== Code Execution Successful ===
```

Table 5-2. Queues using Linked List Implementation

Using Arrays

C/C++

```
#include <iostream>
using namespace std;

class Queue {
private:
	int *arr;
	int front;
	int rear;
	int capacity;
	int size;

public:
```

```
Queue(int cap) {
capacity = cap;
arr = new int[capacity];
front = 0;
rear = -1;
size = 0;
}

~Queue() {
delete[] arr;
}

void enqueue(int element) {
if (isFull()) {
       cout << "Queue is full! Cannot enqueue " << element << endl;
       return;
}

rear = (rear + 1) % capacity;
arr[rear] = element;
size++;
cout << element << " added to the queue." << endl;
}

void dequeue() {
if (isEmpty()) {
       cout << "Queue is empty! Cannot dequeue." << endl;
       return;
}

cout << arr[front] << " removed from the queue." << endl;
front = (front + 1) % capacity;
size--;
}

int peek() const {
if (isEmpty()) {
       cout << "Queue is empty!" << endl;
       return -1;
}
return arr[front];
}

bool isEmpty() const {
return size == 0;
}

bool isFull() const {
return size == capacity;
}

int getSize() const {
return size;
}
```

```
        void display() const {
        if (isEmpty()) {
                cout << "Queue is empty!" << endl;
                return;
        }

        cout << "Queue elements: ";
        for (int i = 0; i < size; i++) {
                cout << arr[(front + i) % capacity] << " ";
        }
        cout << endl;
        }
};

int main() {
        Queue q(5);

        q.enqueue(10);
        q.enqueue(20);
        q.enqueue(30);
        q.enqueue(40);
        q.enqueue(50);

        cout << "Front element: " << q.peek() << endl;
        cout << "Current size: " << q.getSize() << endl;

        q.display();

        q.dequeue();
        q.dequeue();

        cout << "Front element after two dequeues: " << q.peek() << endl;
        cout << "Current size after dequeues: " << q.getSize() << endl;

        q.display();

        q.enqueue(60);
        cout << "Front element after adding 60: " << q.peek() << endl;

        q.display();

        return 0;
}
```

Output

```
/tmp/ubcFN8csfX.o
10 added to the queue.
20 added to the queue.
30 added to the queue.
40 added to the queue.
50 added to the queue.
Front element: 10
Current size: 5
Queue elements: 10 20 30 40 50
10 removed from the queue.
20 removed from the queue.
Front element after two dequeues: 30
Current size after dequeues: 3
Queue elements: 30 40 50
60 added to the queue.
Front element after adding 60: 30
Queue elements: 30 40 50 60


=== Code Execution Successful ===
```

Table 5-3. Queues using Array Implementation

7. Supplementary Activity

1. Create a class called Job (comprising an ID for the job, the name of the user who submitted it, and the number of pages).

```
C/C++
#include <iostream>
#include <string>

using namespace std;

class Job {
public:
	int id;
	string user;
	int pages;

	Job(int jobId, const string& userName, int numPages)
	: id(jobId), user(userName), pages(numPages) {}
};
```

2. Create a class called Printer. This will provide an interface to add new jobs and process all the jobs added so far.

```
C/C++
class Printer {
private:
	static const int MAX_JOBS = 10;
	Job* jobQueue[MAX_JOBS];
	int front;
	int rear;
	int count;

public:
	Printer() : front(0), rear(0), count(0) {}

	bool addJob(Job* job) {
	if (count == MAX_JOBS) {
		cout << "Printer queue is full!" << endl;
		return false;
	}
	jobQueue[rear] = job;
	rear = (rear + 1) % MAX_JOBS;
	count++;
	cout << "Job added: ID " << job->id << " by " << job->user << " (" << job->pages <<
" pages)" << endl;
	return true;
	}

	void processJobs() {
	while (count > 0) {
		Job* job = jobQueue[front];
		cout << "Processing job: ID " << job->id << " by " << job->user << " (" <<
job->pages << " pages)" << endl;
		front = (front + 1) % MAX_JOBS;
		count--;
		delete job;
	}
	cout << "All jobs processed." << endl;
	}
};
```

3. To implement the printer class, it will need to store all the pending jobs. We'll implement a very basic strategy – first come, first served. Whoever submits the job first will be the first to get the job done.

```
C/C++
int main() {
	Printer printer;

	printer.addJob(new Job(1, "Alice", 5));
	printer.addJob(new Job(2, "Bob", 3));
	printer.addJob(new Job(3, "Charlie", 8));

	printer.processJobs();
```

```
        return 0;
}
```

4. Finally, simulate a scenario where multiple people are adding jobs to the printer, and the printer is processing them one by one.

```
C/C++
int main() {
    Printer printer;

    printer.addJob(new Job(1, "Earl", 5));
    printer.addJob(new Job(2, "Nyko", 3));
    printer.addJob(new Job(3, "Paul", 8));

    printer.processJobs();

    return 0;
}
```

Output

```
/tmp/R23f4FS689.o
Job added: ID 1 by Earl (5 pages)
Job added: ID 2 by Nyko (3 pages)
Job added: ID 3 by Paul (8 pages)
Processing job: ID 1 by Earl (5 pages)
Processing job: ID 2 by Nyko (3 pages)
Processing job: ID 3 by Paul (8 pages)
All jobs processed.


=== Code Execution Successful ===
```

5. Defend your choice of internal representation: Why did you use arrays or linked lists?

- I used arrays for making this program because in scenarios like the printing queue, where the maximum number of jobs can be reasonably estimated, using an array provides an efficient, simple, and effective solution. The advantages of using arrays align well with the program's requirements, allowing for clear and maintainable code while optimizing performance and memory management.

8. Conclusion

- In conclusion, The C++ implementation of a printing queue using arrays effectively demonstrates queue management principles, specifically employing a first-come, first-served strategy. By utilizing distinct classes for Job and Printer, the program achieves clear organization and maintainability. The array structure offers advantages such as simplicity, efficient access, and predictable memory usage, making it suitable for scenarios with a known maximum number of jobs. The simulation of job submissions and processing illustrates how

multiple users can interact with a single printing resource, providing a solid foundation for understanding queue operations and efficient data structure usage in software development.