Activity No. <n>	
<Replace with Title>	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:10/7/24**
Section:CPE21S1	**Date Submitted:10/7/24**
Name(s): Jimenez, Christian Joros R.	**Instructor: Ma'am Sayo**

6. Output

Table 5-1. Queues using C++ STL

```cpp
#include <iostream>
#include <queue>
#include <string>

using namespace std;

int main() {
  queue<string> studentQueue;

  studentQueue.push("C++");
  studentQueue.push("Python");
  studentQueue.push("Java");


  cout << "Initial Queue: ";
  while (!studentQueue.empty()) {
    cout << studentQueue.front() << " ";
    studentQueue.pop();
  }
  cout << endl;

  studentQueue.push("Christian Joros");
  studentQueue.push("Ma'am Sayo");

  cout << "Queue after adding two students: ";
  while (!studentQueue.empty()) {
    cout << studentQueue.front() << " ";
    studentQueue.pop();
  }
  cout << endl;

  cout << "is queue empty or not? " << studentQueue.empty() << endl;

  studentQueue.push("Bjorn");

  cout << "Queue after adding one more student: ";
  while (!studentQueue.empty()) {
    cout << studentQueue.front() << " ";
    studentQueue.pop();
  }
  cout << endl;

  return 0;
}
```

```
Initial Queue: C++ Python Java
Queue after adding two students: Christian Joros Ma'am Sayo
is queue empty or not? 1
Queue after adding one more student: Bjorn
```

Explanation:
First-In, First-Out, or FIFO: This is how the queue operates. Push() is used to add elements to the back, and pop() is used to remove elements from the front. front() method: By using the front() method, you can access the element at the front of the queue without removing it. This allows you to examine the following element in line without affecting the queue's current state. empty() method: Before attempting to remove or access the front element, it is helpful to see if the queue contains any elements.

Table 5-2. Queues using Linked List Implementation

```
#include <iostream>
#include <queue>
#include <string>

using namespace std;

int main() {
  queue<string> studentQueue;
  string studentName;

  cout << "1. Inserting an item into a non-empty queue:\n";
  cout << "Enter first item: ";
  cin >> studentName;
  studentQueue.push(studentName);
  cout << "Enter second item: ";
  cin >> studentName;
  studentQueue.push(studentName);
  cout << "Queue after adding two items: ";
  while (!studentQueue.empty()) {
          cout << studentQueue.front() << " ";
          studentQueue.pop();
  }
  cout << endl;

  cout << "\n2. Inserting an item into an empty queue:\n";
  cout << "Enter first item: ";
  cin >> studentName;
  studentQueue.push(studentName);
  cout << "Enter second item: ";
  while (!studentQueue.empty()) {
          cout << studentQueue.front() << " ";
          studentQueue.pop();
  }
  cout << endl;

  cout << "\n3. Deleting an item from a queue of more than one item:\n";
  cout << "Enter first item: ";
  cin >> studentName;
  studentQueue.push(studentName);
  cout << "Enter item to delete: ";
  cin >> studentName;
  studentQueue.push(studentName);
  cout << "Queue before deletion: ";
  while (!studentQueue.empty()) {
          cout << studentQueue.front() << " ";
          studentQueue.pop();
  }
  cout << endl;

  cout << "Enter first item: ";
  cin >> studentName;
  studentQueue.push(studentName);
  cout << "Enter second item: ";
  cin >> studentName;
  studentQueue.push(studentName);

  cout << "Queue after deletion: ";
```

```
  studentQueue.pop();
  while (!studentQueue.empty()) {
        cout << studentQueue.front() << " ";
        studentQueue.pop();
  }
  cout << endl;

  cout << "\n4. Deleting an item from a queue with one item:\n";
  cout << "Enter the name of the student: ";
  cin >> studentName;
  studentQueue.push(studentName);
  cout << "Queue before deletion: ";
  while (!studentQueue.empty()) {
        cout << studentQueue.front() << " ";
        studentQueue.pop();
  }
  cout << endl;
  cout << "Enter item to delete: ";
  cin >> studentName;
  studentQueue.push(studentName);
  cout << "Queue after deletion: ";
  studentQueue.pop();
  while (!studentQueue.empty()) {
        cout << studentQueue.front() << " ";
        studentQueue.pop();
  }
  cout << endl;

  return 0;
}
```

```
1. Inserting an item into a non-empty queue:
Enter first item: josh
Enter second item: darius
Queue after adding two items: josh darius

2. Inserting an item into an empty queue:
Enter first item: garen
Enter second item: garen

3. Deleting an item from a queue of more than one item:
Enter first item: reiko
Enter item to delete: gpu
Queue before deletion: reiko gpu
Enter first item: reiko
Enter second item: gpu
Queue after deletion: gpu

4. Deleting an item from a queue with one item:
Enter the name of the student: iphone
Queue before deletion: iphone
Enter item to delete: iphone
Queue after deletion:
```

Explanation:

1. Inserting an item into a non-empty queue:

- We add two students ("Alice" and "Bob") to a non-empty queue.
- The `push()` method adds the new elements to the back of the queue.

2. Inserting an item into an empty queue:
    - We add one student ("Charlie") to an empty queue.
    - Again, the `push()` method adds the element to the back.
3. Deleting an item from a queue of more than one item:
    - We add two students ("David" and "Emily") to the queue.
    - Then, we use `pop()` to delete the first element from the queue.
    - The `pop()` method removes the element from the front.
4. Deleting an item from a queue with one item:
    - We add one student ("Frank") to the queue.
    - We then use `pop()` to delete the only element in the queue.
    - The `pop()` method will remove the element at the front, leaving the queue empty.

Table 5-3. Queues using Array Implementation

```
#include <iostream>
#include <string>

using namespace std;

template <typename T>
class Queue {
private:
  T* q_array;
  int q_size;
  int q_front;
  int q_back;
  int q_capacity;

  void resize(int new_capacity) {
        T* temp = new T[new_capacity];
        for (int i = 0; i < q_size; ++i) {
        temp[i] = q_array[((q_front + i) % q_capacity)];
        }
        delete[] q_array;
        q_array = temp;
        q_capacity = new_capacity;
        q_front = 0;
        q_back = q_size;
  }

public:
  Queue() : q_size(0), q_front(0), q_back(0), q_capacity(10) {
        q_array = new T[q_capacity];
  }

  Queue(const Queue& other) : q_size(other.q_size), q_front(other.q_front), q_back(other.q_back),
q_capacity(other.q_capacity) {
        q_array = new T[q_capacity];
        for (int i = 0; i < q_size; ++i) {
        q_array[i] = other.q_array[((other.q_front + i) % other.q_capacity)];
        }
  }

  Queue& operator=(const Queue& other) {
```

```
        if (this != &other) {
        delete[] q_array;
        q_size = other.q_size;
        q_front = other.q_front;
        q_back = other.q_back;
        q_capacity = other.q_capacity;
        q_array = new T[q_capacity];
        for (int i = 0; i < q_size; ++i) {
        q_array[i] = other.q_array[((other.q_front + i) % other.q_capacity)];
        }
        }
        return *this;
}

~Queue() {
        delete[] q_array;
}

bool empty() const {
        return q_size == 0;
}

int size() const {
        return q_size;
}

void clear() {
        delete[] q_array;
        q_size = 0;
        q_front = 0;
        q_back = 0;
        q_capacity = 10;
        q_array = new T[q_capacity];
}

T& front() const {
        if (empty()) {
        throw runtime_error("Queue is empty");
        }
        return q_array[q_front];
}

T& back() const {
        if (empty()) {
        throw runtime_error("Queue is empty");
        }
        return q_array[((q_front + q_size - 1) % q_capacity)];
}

void enqueue(const T& value) {
        if (q_size == q_capacity) {
        resize(q_capacity * 2);
        }
        q_array[q_back] = value;
        q_back = (q_back + 1) % q_capacity;
        q_size++;
}

void dequeue() {
```

```
        if (empty()) {
        throw runtime_error("Queue is empty");
        }
        q_front = (q_front + 1) % q_capacity;
        q_size--;
  }
};

int main() {
  Queue<string> studentQueue;

  studentQueue.enqueue("Alice");
  studentQueue.enqueue("Bob");
  studentQueue.enqueue("Charlie");

  cout << "Queue: ";
  while (!studentQueue.empty()) {
        cout << studentQueue.front() << " ";
        studentQueue.dequeue();
  }
  cout << endl;

  return 0;
}
```

```
Queue: Alice Bob Charlie
```

Explanation:

Implementation of a Circular Array: The array's queue elements are managed in a circular manner by the modulo operator (%), q_front, and q_back, which facilitates effective insertions and deletions.

Dynamic Resizing: When the queue fills up, the resize() function takes care of it and makes sure it can expand as needed.

Memory Management: To prevent memory leaks, the code carefully manages memory allocation and deallocation.

A simple queue with the majority of common member functions is implemented by this code. It can be further enhanced by adding new features or modifying its behavior to suit your own requirements.

7. Supplementary Activity

Job Class:

```cpp
#include <iostream>
#include <string>

using namespace std;

class Job {
private:
    int jobID;
    string userName;
    int numPages;

public:
    Job(int id, const string& name, int pages) :
jobID(id), userName(name), numPages(pages) {}

    int getID() const { return jobID; }
    string getUserName() const { return userName; }
    int getNumPages() const { return numPages; }
};
```

Analysis: The Job class encapsulates information about each printing job, including its ID, the user who submitted it, and the number of pages.

Printer Class (Linked List Implementation):

```cpp
#include <iostream>
#include <string>

using namespace std;

class Job;

class Printer {
private:
    struct Node {
        Job* job;
        Node* next;

        Node(Job* j) : job(j), next(nullptr) {}
    };

    Node* front;
    Node* back;

public:
    Printer() : front(nullptr), back(nullptr) {}

    void addJob(Job* job) {
        Node* newNode = new Node(job);
        if (front == nullptr) {
            front = newNode;
            back = newNode;
        } else {
            back->next = newNode;
            back = newNode;
        }
    }

    void processNextJob() {
        if (front == nullptr) {
            cout << "No jobs in the queue.\n";
            return;
        }

        cout << "Processing Job ID: " << front->job->getID()
             << " for user: " << front->job->getUserName()
             << " (Pages: " << front->job->getNumPages() << ")\n";

        Node* temp = front;
        front = front->next;
        delete temp;

        if (front == nullptr) {
            back = nullptr;
        }
    }
};
```

Analysis:

- The `Printer` class uses a linked list to store pending jobs.
- `front` points to the first job in the queue, and `back` points to the last.
- `addJob` adds a new `Job` to the end of the queue.
- `processNextJob` simulates printing the job at the front of the queue and then removes it.

Simulation:

```
#include <iostream>
#include <string>

using namespace std;

class Job {
private:
	int jobID;
	string userName;
	int numPages;

public:
	Job(int id, const string& name, int pages) : jobID(id), userName(name), numPages(pages)
{}

	int getID() const { return jobID; }
	string getUserName() const { return userName; }
	int getNumPages() const { return numPages; }
};

class Printer {
private:
	struct Node {
	Job* job;
	Node* next;

	Node(Job* j) : job(j), next(nullptr) {}
	};

	Node* front;
	Node* back;

public:
	Printer() : front(nullptr), back(nullptr) {}
```

```
        void addJob(Job* job) {
        Node* newNode = new Node(job);
        if (front == nullptr) {
        front = newNode;
        back = newNode;
        } else {
        back->next = newNode;
        back = newNode;
        }
        }

        void processNextJob() {
        if (front == nullptr) {
        cout << "No jobs in the queue.\n";
        return;
        }

        cout << "Processing Job ID: " << front->job->getID()
                << " for user: " << front->job->getUserName()
                << " (Pages: " << front->job->getNumPages() << ")\n";

        Node* temp = front;
        front = front->next;

        delete temp->job;
        delete temp;

        if (front == nullptr) {
        back = nullptr;
        }
        }
};

int main() {
        Printer printer;

        printer.addJob(new Job(1, "Ma'am Sayo", 9));
        printer.addJob(new Job(2, "Ma'am Venal", 21));
        printer.addJob(new Job(3, "Dean", 5));

        printer.processNextJob();
```

```
    printer.processNextJob();
    printer.processNextJob();

    printer.addJob(new Job(4, "Ma'am Escanan", 12));
    printer.addJob(new Job(5, "Ma'am Geneta", 18));

    printer.processNextJob();
    printer.processNextJob();
    printer.processNextJob();

    return 0;
}
```

```
Processing Job ID: 1 for user: Alice (Pages: 10)
Processing Job ID: 2 for user: Bob (Pages: 5)
Processing Job ID: 3 for user: Charlie (Pages: 20)
Processing Job ID: 4 for user: David (Pages: 15)
Processing Job ID: 5 for user: Emily (Pages: 8)
No jobs in the queue.
```

Analysis:

Multiple users adding jobs to the printer is simulated by the code. Jobs are processed by the printer according to first-come, first-served policy. The processNextJob function prints "No jobs in the queue" when the queue is empty.

I chose a linked list for this scenario because the number of jobs in the printer queue can vary. A linked list allows us to add and remove jobs dynamically without needing to pre-allocate a fixed-size array. Linked lists can grow or shrink as needed, unlike arrays, which require a fixed size.

8. Conclusion

Provide the following:

- Summary of lessons learned
  I learned about Queue STL and its application in solving problems. I was able to hone my skills in using queues and linked lists in different scenarios that were presented in this activity.
- Analysis of the procedure
  The procedure mostly used queues in solving problems as well as linked lists that I was able to do with out any problems.
- Analysis of the supplementary activity
  This activity required me to develop a simulation code wherein all computers in an office is only connected to one printer. This activity was pretty interesting to me and I was able to accomplish the requirements that were needed for this problem.
- Concluding statement / Feedback: How well did you think you did in this activity? What

are your areas for improvement? In my opinion, I think I did okay for this activity, I was able to finish this activity in time without any major problems that would have hindered my process in finishing the activity
9. Assessment Rubric