CSC 222: Computer Programming II

Spring 2004

Queues and simulation
- queue ADT
  - enqueue (push), dequeue (pop), front, empty, size
- <queue> library
- application: bank simulation
- OOP design, testing strategies

Lists & queues

recall: a *stack* is a simplified list
- add/delete/inspect at one end
- useful in many real-word situations (e.g., delimiter matching, run-time stack)

queue ADT
- a *queue* is another kind of simplified list
- add at one end (the back), delete/inspect at other end (the front)

DATA: sequence of items
OPERATIONS: enqueue (push at back), dequeue (pop off front), look at front,
                 check if empty, get size

these are the ONLY operations allowed on a queue
   — queues are useful because they are simple, easy to understand
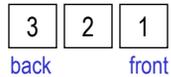   — each operation is O(1)

Queue examples

- line at bank, bus stop, grocery store, …
- printer jobs
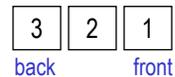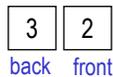- CPU processes
- voice mail

a queue is also known as
- first-in-first-out (FIFO) list
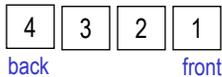
3	2	1

back   front

enqueue (push): adds item at the back

4	3	2	1

back   front

deueue (pop): removes item at the front

3	2

back   front

front: returns item at the front → 1

3	2	1

back   front

3

Queue exercise

- start with empty queue
- ENQUEUE 1
- ENQUEUE 2
- ENQUEUE 3
- FRONT
- ENQUEUE 4
- DEQUEUE
- DEQUEUE
- FRONT
- ENQUEUE 5

4

<queue> library

since a queue is a common data structure, a predefined C++ library exists

```
#include <queue>
```

the standard `queue` class has the following member functions

```
void push(const TYPE & item);
void pop();
TYPE & front() const;
bool empty() const;
int size() const;
```

```
queue<int> numQ;

int num;
while (cin >> num) {
    numQ.push(num);
}

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

```
queue<int> numQ1, numQ2;

int num;
while (cin >> num) {
    numQ1.push(num);
}

while ( !numQ1.empty() ) {
    numQ2.push( numQ1.front() );
    numQ1.pop();
}

while ( !numQ2.empty() ) {
    cout << numQ2.front() << endl;
    numQ2.pop();
}
```

Queues and simulation

queues are especially useful for simulating events

e.g., consider simulating a small-town bank
- 1 teller, customers are served on a first-come-first-served basis
- at any given minute of the day, there is a constant probability of a customer arriving
- the length of a customer's transaction is a random int between 1 and some MAX

```
What is the time duration (in minutes) to be simulated? 10
What percentage of the time (0-100) does a customer arrive? 30

2: Adding customer 1 (job length = 4)
2:   Serving customer 1 (finish at 6)
4: Adding customer 2 (job length = 3)
5: Adding customer 3 (job length = 1)
6:     Finished customer 1
6:   Serving customer 2 (finish at 9)
9: Adding customer 4 (job length = 3)
9:     Finished customer 2
9:   Serving customer 3 (finish at 10)
10:     Finished customer 3
10:   Serving customer 4 (finish at 13)
13:     Finished customer 4
```

OOP design of bank simulation

what are the objects/actors involved in the bank simulation?

Customer
- get the customer's ID
- get the customer's arrival time
- get the customer's job length

Teller
- get the teller's ID
- get the teller's status (busy? free? finished?)
- start serving a customer
- get information on the customer being served

ServiceCenter
- add a customer to the queue
- check to see if any customers waiting
- check to see if a customer is being served
- perform the next step (serve customer if available)

7

Bank simulation classes

given this general scheme, can design the interfaces for the classes

will worry about the data fields & implementation details later

```
class Customer {
  public:
    Customer(int id = 0, int arrTime = 0);
    int getID();
    int getArrivalTime();
    int getJobLength();
  private:
    ???
};
```

```
enum Status { FREE, BUSY, FINISHED };

class Teller {
  public:
    Teller(int id = 0);
    int getID();
    Status getStatus(int time);
    void serveCustomer(Customer c, int time);
    Customer getCurrentCustomer(int time);
  private:
    ???
};
```

```
class ServiceCenter
{
  public:
    ServiceCenter();
    void addCustomer();
    bool customersWaiting();
    bool customersBeingServed();
    void serveCustomers();
  private:
    ???
};
```

8

```
#include "ServiceCenter.h"
#include "Die.h"

bool CustomerArrived(int arrivalProb);

void main()
{
  int maxTime, arrivalProb;
  cout << "What is the time duration (in minutes) to be simulated? ";
  cin >> maxTime;
  cout << "What percentage of the time (0-100) does a customer arrive? ";
  cin >> arrivalProb;
  cout << endl;

  ServiceCenter bank;
  for (int time = 1; time <= maxTime
                      || bank.customersBeingServed()
                      || bank.customersWaiting(); time++) {
      if ( time <= maxTime && CustomerArrived(arrivalProb) ) {
          bank.addCustomer();
      }
      bank.serveCustomers();
  }
}

/////////////////////////////////////////////////////////////////////

bool CustomerArrived(int arrivalProb)
// Assumes: 0 <= arrivalProb <= 100
// Returns: true with probability ARRIVAL_PROB, else false
{
  Die d(100);
  return (d.Roll() <= arrivalProb);
}
```

9

Customer class

what data fields are needed to implement the member functions?

let's do it!

```
class Customer {
  public:
    Customer(int id = 0, int arrTime = 0);
    int getID();
    int getArrivalTime();
    int getJobLength();
  private:

    static const int MAX_LENGTH = 8;
    int customerID;
    int arrivalTime;
    int jobLength;

};
```

how do we test the class?

- don't wait until all the classes are implemented and test them all together

```
#include <iostream>
#include "Customer.h"
using namespace std;

void main()
{
  for (int i = 0; i < 10; i++) {
    Customer c(100+i, i);
    cout << "ID: " << c.getID() << endl
         << "arrival time: " << c.getArrivalTime() << endl
         << "job length: " << c.getJobLength() << endl << endl;
  }

  return 0;
}
```

10

Teller class

what data fields are needed to implement the member functions?

let's do it!   testing?

```cpp
enum Status { FREE, BUSY, FINISHED };

class Teller {
  public:
    Teller(int id = 0);
    int getID();
    Status getStatus(int time);
    void serveCustomer(Customer c, int time);
    Customer getCurrentCustomer(int time);
  private:

    int tellerID;
    int busyUntil;
    Customer serving;

};
```

```cpp
#include <iostream>
#include "Teller.h"
using namespace std;

void main()
{
  Teller t(1234);
  cout << "ID: " << t.getID() << endl
       << "Status: " << t.getStatus(1) << " (0=FREE,1=BUSY,2=FINISHED)" << endl;

  Customer c(999, 3);
  t.serveCustomer(c, 5);
  for (int i = 5; i < 14; i++) {
    cout << "Status at time " << i << ": " << t.getStatus(i) << endl;
  }

  cout << "Customer at time 5: " << t.getCurrentCustomer(5).getID() << endl;

  return 0;
}
```

11

ServiceCenter class

what data fields are needed to implement the member functions?

```cpp
class ServiceCenter
{
  public:
    ServiceCenter();
    void addCustomer();
    bool customersWaiting();
    bool customersBeingServed();
    void serveCustomers();
  private:

    queue<Customer> waiting;
    Teller loneTeller;
    int currentTime;
    int numEntered;

};
```

HW5: implement this class
- constructor initializes the data fields

- `addCustomer` adds a new customer to the waiting queue (& display a message)
  customer ID is # entered (1st customer = 1, 2nd customer = 2, …)
  arrival time is the current time, as maintained by the class

- `customerWaiting` returns true if any customers waiting in queue (not being served)
- `customerBeingServed` returns true if a customer is currently being served

- `serveCustomers` does one step in the simulation
  if the teller finishes with a customer, remove them (& display a message)
  if the teller is free and there is a customer waiting, start serving them (& display a message)
  increment the current time

12