Taxi dispatch

- Recap:
  - Customers arrive and have to be assigned to (earliest waiting) taxies.
- We see next how to implement the queue using member functions

A struct to represent the queue

- N = max number of waiting taxis
- ***nwaiting*** = number of taxis currently waiting.
- ***front*** = index of earliest taxi
- Array elements ***front .. front+nwaiting%N*** hold the ids of waiting taxis.
- The queue is involved in two operations: inserting taxis and removing taxis.
- These become member functions.
- It is useful to have a member function to initialize as well.

```
const int N=100;
struct Queue{
 int elements[N],
    nwaiting,front;
 void initialize(){..}
 bool insert(int v){
  ...
  }
 bool remove(int &v){
  ...
 }
};
```

Main program

```
int main(){
 Queue q;
 q.initialize();
 while(true){
  char c; cin >> c;
  if(c == 'd'){
    int driver; cin >> driver;
    if(!q.insert(driver)) cout <<"Q is full\n";
  }
  else if(c == 'c'){
    int driver;
    if(!q.remove(driver)) cout <<"No taxi available.\n";
    else cout <<"Assigning <<driver<< endl;
  }
}
```

Member function initialize

```
struct Queue{
  ...
  void initialize()}
    nWaiting = 0;
    front = 0;
  }
};
```

- We were doing this at the beginning in the old program.
- It now happens through this member function.

Member function insert

```
struct Queue{
  ...
  bool insert(int v){
    if(nWaiting >= n)
      return false;
    elements[(front +
      nWaiting)%N] = v;
    nWaiting++;
    return true;
  }
};
```

- A value can be inserted only if the queue has space.
- The value must be inserted into the next empty index in the queue.
- The number of waiting elements in the queue is updated.
- Return value indicates whether operation was successful.

Member function remove

```
struct Queue{
  ...
  bool remove(int &v){
    if(nWaiting == 0)
      return false;
    v = elements[front];
    front = (front+1)%N;
    nWaiting--;
    return true;
  }
};
```

- A value can be removed only if the queue contains some values.
- The value must be from the front of the queue.
- The value removed is returned in the reference parameter v.
- Update ***front*** and ***nWaiting***.
- The return value of the function denotes whether the operation was successful, i.e. whether something is returned.

Exercise

- Add a member function which checks if a certain driverID is waiting, and if so, returns how many driverIDs are before it.
  - Return a struct with a bool member saying whether the driverID is among those waiting, and an int member giving the position.

Remarks

- The member functions only contain the logic of how to manage the queue.
  - We had identified invariants about nWaiting, front etc.
  - These apply only to the member functions
- The main program only contains the logic of dealing with taxis and customers.
- The new program has become simpler as compared to Chapter 14, where the above two concerns were mixed up together.

Concluding remarks

- Structures should be used to collect together the attributes of an entity and generally represent an entity.
- Member functions should be written to represent valid operations/actions of the entities.
- The invariants we define for the entities should be satisfied by our functions.
- Later we will see ways by which we can prevent accesses other than those through member functions.
  - Useful in persuading ourselves that we are not wrongly accessing a structure “even by mistake”.