An Introduction to Programming through C++
Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Lecture No. 20 Part- 3
Structures Part 2
Taxi Dispatch

Welcome back.

(Refer Slide Time: 0:26)

What we discussed

- Syntax of member functions
- A member function is physically placed inside the struct definition.
- A call to a member function looks like:

```
receiver.function-name(...)
```

- The member function executes like an ordinary function, with receiver being an additional argument.
  - The syntax highlights the special relationship of the function to the receiver.
  - Receiver is implicitly passed by reference, i.e. the call can modify the members in the receiver.
- The benefits of this syntax will become clear soon.

In the last segment we discussed member functions. And in this segment, we will take the discussion further and see it in the context of the taxi dispatch problem.

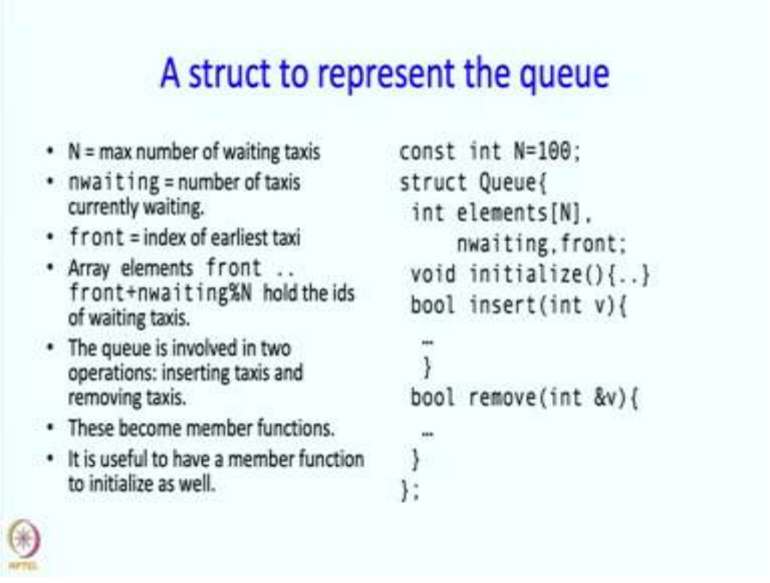
(Refer Slide Time: 0:21)

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

So, let me remind you what the taxi dispatch problem was. So, customers arrive and have to be assigned to earliest waiting taxi. And, we now are going to implement the functionality which we implemented earlier using member functions.

(Refer Slide Time: 0:53)



So first of all, a structure to represent the queue, okay. So, let us say N is the maximum number of waiting taxis and waiting is the variable which we are going to need to keep track of the number of taxis currently waiting. Front is the index of the earliest taxi in the array that we are going to keep. And the array elements front through front plus and waiting mod N hold the IDs of the waiting taxis.

So remember that in chapter 14, we discussed this and there we said that this is a circular queue that is why that mod N operation happens. N is the length of that queue, maximum number. of waiting taxes possible. And the queue is involved in two operations, inserting taxis and removing taxis, so these become natural member functions. And you will see that it is useful to have a member function to initialize the queue as well. So, here is what the queue might look like. So we have constant at N=100, so we need to have some maximum length or the length of the queue and so that is 100. Then we have array elements, which are going to store which are going to store the driver IDs of length m and then we are also going to have variables members which are n waiting and front. Then these are our three member functions. So, that is that is sort of the overall high level scheme of what we are going to do in this taxi dispatch problem.

So, let us just let me just show you what each of these functions is going to look like. So, let us start with initialize. So what do we want in initialization? If you remember when we created the queue, we are we said that n waiting and front should become 0 and that is kind of a bare minimum that we want the queue that we create should, I mean it should make sense. So n waiting and front should make sense and at the beginning that is how they make sense. So, that is what we are going to do in this definition. So, insert is going to contain the code of what is needed, what is needed to happen when you insert an additional element into the queue and remove is going to contain the code which is needed for removing elements from the queue.

(Refer Slide Time: 3:31)

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;
  }
```

So, what does our main program look like? So, we create a queue, the struct, and then we initialize it or there should be parentheses there, so that is a typo, we need to have parenthesis and a semicolon over here. And then there is a loop, so we read in the command which the operator gives and if it is d then we are going to get the driver details and we are going to try and insert it into the array into the queue.

If it is a c for a customer, then again, we are going to remove; we are going to remove the driver. So, we are going to take out the driver from the queue and this is going to be a reference argument, so this will actually modify this driver. And, of course, in either case, in this case, if there is no driver waiting, then we should print this message.

In this case, if there is no space to put in the new driver we are also going to put this message. And if that driver removal is successful then we are going to say print a message saying

'Assigning' this driver. That is basically the main program, so the member function initialized. So, this is actually fairly simple as I just described.

(Refer Slide Time: 5:02)

Member function initialize

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

- We were doing this at the beginning in the old program.

It is going to simply set n waiting to 0 and front also to 0. So, we were doing this at the beginning in the old program but now notice that it has gone into a member function.

(Refer Slide Time: 5:17)

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;
  }
```

So, in particular, if you look at this the internals of queue are not really visible and that is a good thing, in the sense that this code is a high-level code. The internals are being manipulated but they are being manipulated in these calls in the member function calls. So, this is the kind of separation of concerns that has come about because of all of this. So in

particular, the member functions modify the members of queue and the user which is this main program in this case is just going to call the appropriate member functions.

And the user where the main program is not really aware even of what happens inside this queue. If you remember we ourselves had two implementations of the queue and this main program would work with either implementation, provided the functions themselves were changed. But this program would not have to change, if the signature of the member functions does not change.

(Refer Slide Time: 6:23)

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.

The member function insert is again not hard to write. So, we are inserting a driver v and we are going to check, if the queue has space. And if so we are going to insert it at the front and we had a fairly extensive discussion of where, we had a fairly extensive discussion of where the back of the queue is. So, it is a disposition, so that is where we insert. And then the increment and weighting and we return true, because this is successful. If there are too many people waiting, then the insertion is not successful and so we return false.

So member functions remove is somewhat similar, so we check whether the queue contains some values, otherwise we return false. Then the value to be removed should come from the front of the queue. So we take the value from the front, and we increment front but of course this increment must be this circular increment that is, if we are already at the last element of the queue, then we should go to go back to the zeroth element okay. And of course any waiting should decrease okay.

(Refer Slide Time: 7:48)

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.

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;
  }
}
```

So so as you can see we can, in fact, easily write taxi dispatch in terms of member functions. And one point to note and I am repeating it, but it is important which is that in this entire program we are not really looking at the details of the queue. So our concerns have been nicely separated, the member functions worry about the details of the queue and the user program just calls the write functions. So, this exercise is about adding one more operation so you should certainly attempt it.

(Refer Slide Time: 8:30)

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.

So, the member functions only contain the logic of how to manage the queue. And we had defined invariants about nwaiting front and things like that. And these apply only to the member functions, the main program is not concerned with those invariants. The main program only contains the logic of dealing with the taxis and the customers. So this is the so-called separation of concerns.

And in some sense the new program is simpler as compared to what we had earlier, where these concerns were mixed up. So if you can separate out concerns it is always a good idea. So, that really concludes this lecture and here are some concluding remarks.

(Refer Slide Time: 9:19)

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".

So, structure should be used to collect together the attributes of an entity and generally represent an entity. So if there is an important entity in your program then you should have a

structure which is representing that entity. If that entity is made up of parts, then your structure should contain the parts and the parts themselves may be structures, that is fine. But usually it is good to have this kind of a correspondence.

Member function should be written to represent valid operations and actions of the entities. And the invariants which we define for the entity should be satisfied by our member functions. Now, later on, we will see that we can make this whole philosophy a little bit stricter. In the sense that so far we have been recommending that accesses to entities should go through member functions but later on we will be able to force this view. We would be able to say or we would be able to make the compiler declare an error if the user tries to, he tries to access an entity in some other ways. So you may or may not want to do that, but you should note that if if you sort of design structures or such objects which are strict, then you can you can be sure that look that, I am not doing the wrong thing even by mistake. I am always am always acting according to the rule book and therefore I could not be making a mistake.

And, of course, you should be paranoid about making mistakes because a program is going to be run in so many ways and so many times and you cannot afford to make a mistake in even one of those times. And therefore, anything that reduces mistakes is a good thing. So, that concludes chapter 17 of the book and I definitely will encourage you to solve problems at the end of that chapter and we will stop this lecture. Thank you.