

**An Introduction to Programming through C++**  
**Professor Abhiram G Ranade**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology Bombay**  
**Lecture No. 24: Part - 4**  
**Data structure based programming**  
**Composing data structures**



Welcome back.

(Refer Slide Time: 0:19)

**What we have discussed**

- Implementation of vector and set
- Maps are implemented like set: log time needed for indexing

Next: Composing sets, vectors, maps, strings..



In the previous segment we discussed implementation of the classes, vector and set. And also a little bit about maps. In this segment, I am going to talk about how you use these classes or data structures in programming. And in particular, in programming, you will need to compose them together. And how do we do that.

(Refer Slide Time: 0:39)

## Many applications require composing data structures

Example 1:

- Want to keep track of friends of different people, oldest to youngest

What do we want for a single individual?

- List of friends in oldest to youngest order
- `vector<string>` : string holds name of friend

For all people:

```
map<string, vector<string> > friends;  
friends["Amitabh"].push_back("Dharmendra");  
friends["Amitabh"].push_back("Vinod");  
cout << friends["Amitabh"][0] << endl;  
//will print Dharmendra
```



So, let me take an example, suppose I am writing a program in which I want to keep track of friends of different people, and let us say for each person I want to make a list of his or her friends in the order oldest to youngest. And then maybe I (may) I might want to do some processing on those lists. So, what Data Structure do we use? Well, first of all, what is it that I require this data structures for all people, lots of people. So, let me begin by asking what is it that I want for a single individual?

So, for a single individual I want list of friends in oldest to youngest order. So, naturally, what I can use is a vector of strings. So the string will hold the name of the friend and I have a vector, so I can have, I can put lots of names in it. And I can put the name in the order oldest to youngest. So, this is for one individual. So, if I want it for all what do I want? Well, I should build a map, so for every person whose name is here I want this data structure. So, I should have map.

And let me call that map the friends map. So just to clarify this let me give an example of how we are going to use this map. So, I might say, for example, friends[Amitabh] dot and this will get me all the friends of Amitabh, and on that I am pushing back Dharmendra. So, right, if this is the very first statement then initially this will give me a vector which is empty and into that will be

added Dharmendra. I can do that again, so again what does this do? This gives me, this first part give me all the friends of Amitabh. So what are the friends of Amitabh?

Well 'friends' is a map, so friends of Amitabh, gets me this element. It gets me a vector of Strings and onto that I am pushing back another string which is what I should do so I am pushing back Vinod. So, what does friends[Amitabh] contain right now? Well, it contains Dharmendra followed by Vinod. I could write one more command, friends[Dharmendra].push\_back(Amitabh), for example. So then friends will contain a pair for Amitabh and one pair for Dharmendra. And I can put friends of different people inside this friends map.

And the second element, this map is going to take name and it is going to return a list or a vector of friends names. And to get to that, how do I get to that? Well a vector is going to be indexed by numbers. So, for example, if I can I can now write something like this. Friends[Amitabh][0]. So, give me the zeroth friend of Amitabh. Select again, let us look at what it means, so friends of Amitabh is going to give me a vector. And since this is a vector it is legal to take the zeroth indexed element of it and what is this zeroth index element?

Well the push back happened in this order and therefore, I am going to get the zeroth index element which is Dharmendra, so Dharmendra is going to get printed as a result of this. So, this is one example of composing data structure. So, what have we composed? Well we have composed map and of course there is string and we have vector. So we have made a complicated looking data structure, but if you look at it closely it is actually not that complicated. Because vector of strings is a list of strings or it is a list of my friends.

And then for every person, I am storing such a list. So the trick is you read it inside out. Or I guess you can read it outside in as well. So you can say that look for every string something is being stored over here and what it is.

(Refer Slide Time: 5:10)

## Composing data structures

Example 2:

- Want to keep track of friends
  - Want to determine if A and B are friends
- ```
map<string, set<string> > friends;  
friends["Amitabh"].insert("Dharmendra");  
friends["Amitabh"].insert("Vinod");  
cout << friends["Amitabh"].count("Jeetendra");
```
- will print 0 because not friend.
  - Not easy with vectors
  - Vectors can tell you who is the oldest friend, if you insert in order.



Let me take another example, so again this is about keeping track of friends. But, now I want to determine if A and B are friends. So, I want to determine if Dharmendra and Amitabh are friends or Jeetendra and Dharmendra are friends and things like that. So, I want to have a data structure which can quickly answer queries like this. So, what should I use? Well let me give you the answer. So, here it turns out that is useful to use this a map from string to a set of strings.

So, again I am going to call it friends. So, let me read this, so the first thing is going to be the name of the person whose friends I am thinking of. And then this set is going to contain the set of all the friends. So, how do I use this? So, earlier I pushed back Dharmendra, now I am going to insert. Why insert? Because friends[Amitabh] gets me a set of strings not a vector strings. And into to a set you insert you do not push back.

So, again I am going to insert Vinod as before and now I can query. So, here is how I am going to query. So, I am going to ask give me friends of Amitabh and what does it give me? It gives me a set of friends. And inside that count how many times Jeetendra appears. So, if it appears, if this appears once, then that means yes Jeetendra is a friend. If this appears, if this returns a zero then it means Jeetendra is not a friend.

So, what is going to happen now? Well into the set we inserted Dharmendra and we inserted Vinod and therefore, Jeetendra is not there and therefore, this is going to printout a zero because we have not made Jitendra a friend. But if you insert even now, `friends[Amitabh].insert("Jeetendra")` and then call this again, then you will see that the count has become one. And by the way these are sets. So, if you reinsert it is not going to change anything. The sets as you know from your study of mathematics, a set cannot contain the same element more than once.

So, C++ actually does have something called a multiset, which does contain, which allows you to store several copies of the same element into the set. So as I said, look at the online documentation if you feel you need to learn what a multiset is. But we are not going to look at this interesting class in this course because already we are doing a lot. Now, I just want to point out that this kind of a query to quickly decide whether Jeetendra is a friend of Amitabh would not be easy with vectors. You have to write a little bit more code.

So what code would we write? So we would say look at that vector, compare Jeetendra to every element in that vector. That kind of code we would have to write. But here since it is a set and on the set we have this count operation, we can get the answer in exactly in in just very small amount of code over here. What is the important point that is coming out of this example, the data we are storing is really the same. We are storing information about who is the friend of who, but depending upon what data structure we use certain kinds of queries are easier to answer.

So, here what is easy to answer? Whether or not two people are friends is easy to answer. What is not easy to answer? Well this data structure you cannot tell who is an, who is the oldest friend. On the other hand if you had vectors and if you insert friends in order of how long you have known them, then you could quickly tell who is the oldest friend. Just look at the zeroth friend in that vector. So again this is a very important point, you should know how to compose data structures. But just because you are storing the same information, it does not mean that exactly one kind of data structure is the right data structure. It really depends upon what you want to do with that information.

(Refer Slide Time: 10:08)

## Composing data structures

Example 3:

- Want to represent bus fares between cities
- Want to know fare between city A and city B (assuming there is a bus connection)

What do we need to know for each city?

"What cities can be reached directly, and the respective fares."

= map from reachable city to fare payable

For all cities:

Map from city to (map from reachable city to fare payable)

map<string, map<string, double> > fare;

fare["Mumbai"]["Pune"] = 500; // Mum-Pun fare is 500.



I want to do one more example, so let us say we want to create a fare table. We want to represent bus fares between cities. So once we have the table, we could ask questions like, what is the fare between city A and city B. Assuming of course is stored it in our table in the first place. So if there is a bus, bus connection from city A to city B, and somebody gives us a query, tell me the fare from A to B, we should be able to look at our table and answer this very quickly. How do we do this? Well, let us sort of try to build it bottom up or inside out sort to say.

So, let us ask what do we need to remember for each city, what do we need to remember? Well we need to remember what cities can be reached directly from that city and what fares are there for each of those journeys. So this is what we want to remember for the first city, second city, third city whatever it is. For Mumbai, Pune, Nagpur, Kolhapur whatever cities you are talking about we need to you remember this information.

Well what is this information? So, for every city that can be reached, we want to keep track of the respective fare. So this is really just a map from city to fare payable. So, what kind of map is it going to be? So again let us say the reachable city is represented by strings. So, it is going to be a map from strings to fare, fare could be double. So, it could be a map from strings to double. So this is just for one city. And what else do we, what do we really want? We want this for all cities. So if you want it for all cities, what should we do?

Well we want this is going to be inside something and what is that outer thing? Well we want to map from a city to a map. So, for Mumbai I want a map of all cities reachable and what fares are payable. So it is a map from a city name to such a map. So here is what it is going to look like. So map from a string to a map from a string to a double. So, this is really this part and this map, the city over here is this part. So, I have to supply a city, I have to supply another city and out will come the double or I can store into this.

So, for example, once I declare this I can write a statement like this. So fare from Mumbai to Pune is 500 rupees. That is what this statement is saying. But you should really understand what this is doing. So this is saying there is a variable or there is data structure called fare. What is that? It is a map. So, I can give you an index which is not here to be a number, it could be a string as it is mentioned over here. So, if I give you Mumbai what do I get? I get a map and what is that map? That map is going to tell me, what is the price to go from Mumbai to various cities? So, for every city that is reachable from Mumbai I am going to get the name of the city and the fare. But once I have that map, if I supply where I want to go, I can store into it the fare or if I say print this out, I will get the fare get printed.

(Refer Slide Time: 14:04)

### What we discussed

- Sometimes we need to compose data structures.
- Exact choice of data structures:
  - Depends upon what operations we need to perform.
- How to choose: look at a small part of the entire picture
  - What do we need for one individual?
  - What do we need for a single city?
- Build up from that.

Next: typedef



Alright, so what did we discuss? So, we discussed that sometimes we need to compose data structures. The exact choice of data structures depends on what information we want to store.

But also on what operations we need to perform on that information. And then we also said something about how do you do this composition. So we said that the way we do this composition is look at a small part of the entire picture and build it up from there.

So, what do we need to for one individual? What do we need to do for a single city? Therefore, what do we need to do for all individuals or the all cities together? So, next I am going to talk about something called typedef and then I am going to conclude this lecture sequence. But before that let me take a quick break.