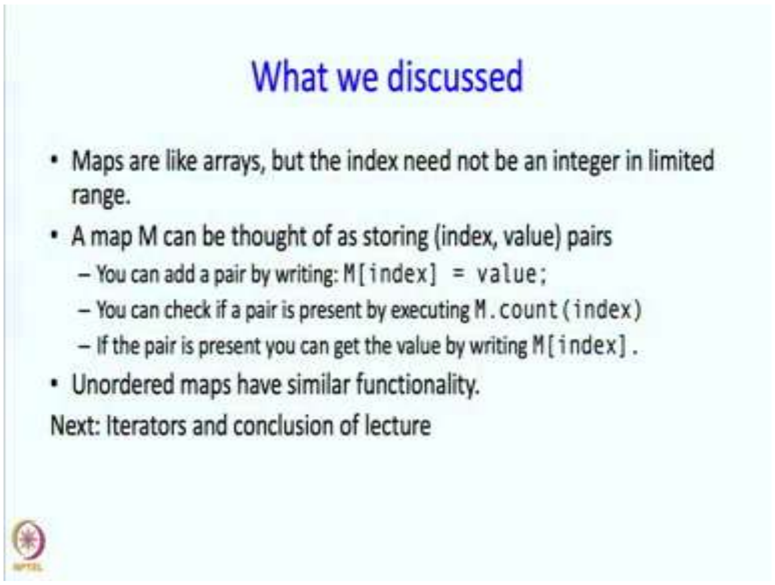
An Introduction to Programming through C++
Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture No. 23 Part – 5
The standard library
Iterators

Welcome back.

(Refer Slide Time: 0:18)



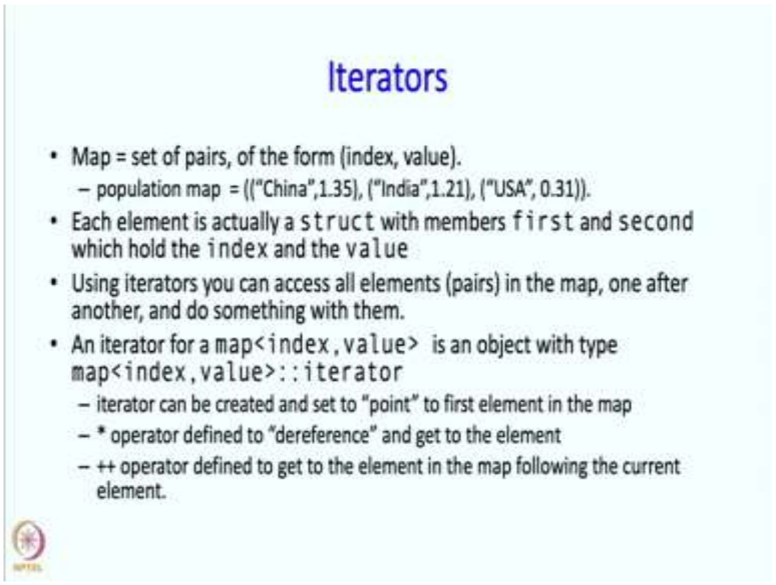
What we discussed

- Maps are like arrays, but the index need not be an integer in limited range.
- A map M can be thought of as storing (index, value) pairs
  - You can add a pair by writing: `M[index] = value;`
  - You can check if a pair is present by executing `M.count(index)`
  - If the pair is present you can get the value by writing `M[index]`.
- Unordered maps have similar functionality.

Next: Iterators and conclusion of lecture

In the previous segment we talked about maps and unordered maps. In this segment we are going to talk about iterators. And we will also conclude this lecture sequence.

(Refer Slide Time: 0:31)



Iterators

- Map = set of pairs, of the form (index, value).
  - population map = (("China",1.35), ("India",1.21), ("USA", 0.31)).
- Each element is actually a `struct` with members `first` and `second` which hold the `index` and the `value`
- Using iterators you can access all elements (pairs) in the map, one after another, and do something with them.
- An iterator for a `map<index,value>` is an object with type `map<index,value>::iterator`
  - iterator can be created and set to "point" to first element in the map
  - * operator defined to "dereference" and get to the element
  - ++ operator defined to get to the element in the map following the current element.

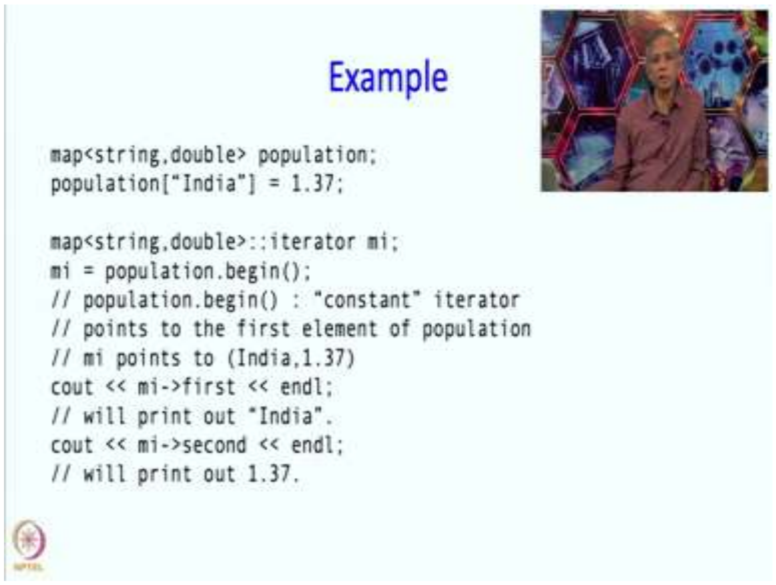
Let me remind you that a map is a set of pairs of the form index and value. Okay, and so for example the population map is this as we define is the sequence. I guess the numbers were slightly different but anyway this could be a population map. Now, it turns out that C++ stores these pairs as a struct. So, each pair is a struct in some structure, in some, yeah, in some container as it is called. And the struct has members first and second.

And first holds the index and second holds the value. So, using iterators you can access all the elements or all these pairs in the map one after another, do what whatever you want with them. So, an iterator from map going from, say index, type index to type value, is an object of type map<index,value>::iterator. So, this iterator also has a type and this type is this complicated looking thing.

All right, so iterators can be created and they can be set to point the first, point to the first element in the map. So, I said earlier that an iterator is like a pointer. So, in that sense when I create an iterator. I can set it to the first element in the map. And then the dereferencing operator, the star operator is defined and you can use it to get then element that this iterator is actually pointing to.

And then the plus operator is also defined and that allows you to go to the next, the next element in the map. So the analogy here is I guess to indices in arrays. So, if you do plus plus to an index you get to the next element of the array. So similarly with here if you do plus plus to an iterator you get to the next element of the map.

(Refer Slide Time: 3:03)



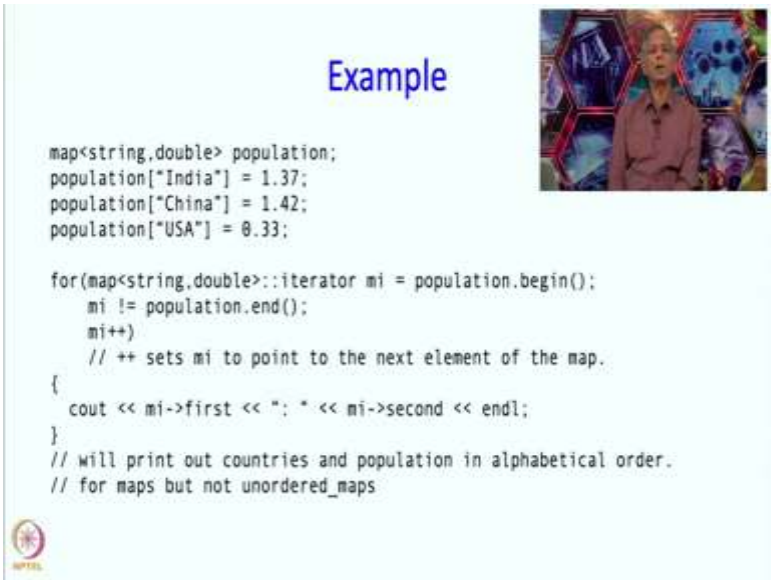
Example

```
map<string,double> population;
population["India"] = 1.37;

map<string,double>::iterator mi;
mi = population.begin();
// population.begin() : "constant" iterator
// points to the first element of population
// mi points to (India,1.37)
cout << mi->first << endl;
// will print out "India".
cout << mi->second << endl;
// will print out 1.37.
```

Okay, so let us do this by as an example. So, our population map, so let us say I store 1.37 as the population of India and now let me create an iterator. So, mi is a variable that I create but of type iterator. It does not have any, it is not assigned any value yet. So, I have to put a value into it. So, I can write mi equal to population.begin(). So, now population.begin() is what might be called a constant iterator. It is like writing mi equal to zero. So this iterator points in a metaphorical sense to the first element or the beginning element in this map.

So, this is a constant operator and it points the first element of population. Well, in this case the first, there is only element so it will point to this element itself. So, if I write cout<<mi->first. So, remember that we have to think of this is a pointer. So, we are first dereferencing it and we are getting to the member first of the dereferenced object. So, of course, that can be written as by the arrow or the minus greater than operator or the arrow operator. And this will print out India and this will print out 1.37.

(Refer Slide Time: 4:34)



Example

```
map<string,double> population;
population["India"] = 1.37;
population["China"] = 1.42;
population["USA"] = 0.33;

for(map<string,double>::iterator mi = population.begin();
    mi != population.end();
    mi++)
    // ++ sets mi to point to the next element of the map.
{
  cout << mi->first << ": " << mi->second << endl;
}
// will print out countries and population in alphabetical order.
// for maps but not unordered_maps
```

But you can do more with this. So, again I have our population map, but let us say we put in all over three countries into it. Now, I have a for loop. And the control variable for the for loop is our iterator and it has been initialized to population dot begin. Next, we are supposed to supply the termination condition for the loop. And this is another constant iterator. So, we are going to do this until so long as this does not equal the end.

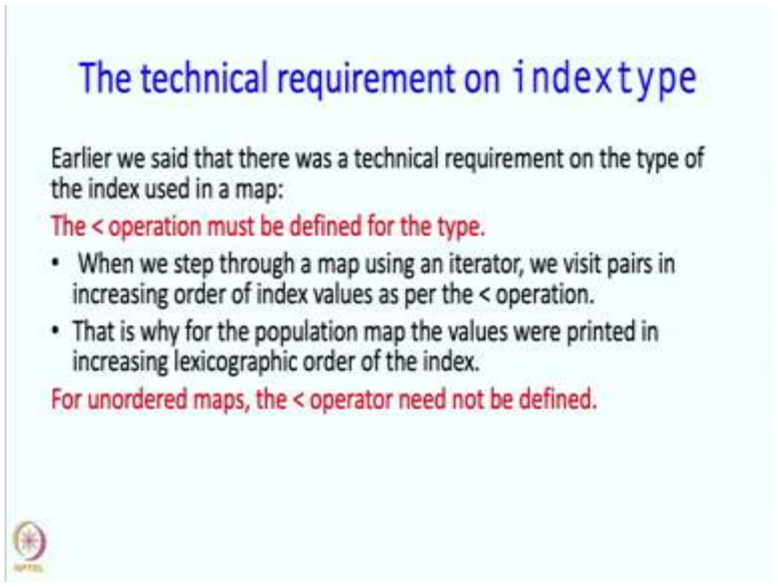
So, you can think of this as an iterator which is just outside the valid iterator ranges. So, again if you go back to the array this is kind of the length of the array. So, you do not want, you do not want your index to become as large as the length. So, that is what this is doing. And you are going to increment that iterator. So, at the end of the loop you are going to increment the iterator so you go on to the next element of the iterator.

So, to start with mi points to the first iterator and then you check whether your iterator is pointing to a valid element. If so, you execute the body and at the end of the body you advanced the iterator. So may be inside the body we are just going to print things out, so whatever the iterators pointing to the first element or the country we will print and then we will put colon and will print the second value, that is it.

So, what should this do. Well, you would think that they should print out the countries and their population and that is indeed what happens. But there is something nice, further nice that happens. So this iterator, this in this case it will print the countries and populations in the alphabetical order. Or the lexicographic order. And I said earlier that maps require a technical, there is a technical condition which is that the less than operator should be defined on the index types and the index types are strings.

And the less than operator on the strings is the lexicographic order. And therefore, when you advance using the iterator the elements are accessed or visited in the lexicographic order. And so things will get printed in lexicographic order. This happens for maps but not for unordered maps. As the name says unordered maps have no order. So, you will get the elements in some order but there may be nothing interesting as far as that order is concerned. That order is something that the implementation decides, it may not even be the element that you inserted first; it will be, it could be some arbitrary order. Again there will be reasons for why it is at ordered but the reasons are not interesting from our point, right now.

(Refer Slide Time: 7:52)



The technical requirement on indextype

Earlier we said that there was a technical requirement on the type of the index used in a map:

The < operation must be defined for the type.

- When we step through a map using an iterator, we visit pairs in increasing order of index values as per the < operation.
- That is why for the population map the values were printed in increasing lexicographic order of the index.

For unordered maps, the < operator need not be defined.

Yeah, so we said that there was a technical requirement and the less than operation. And when we stepped through the map using an iterator we visit the pairs in increasing order. And the less than operator need not be defined but the printing will not happen in it in any interesting order.

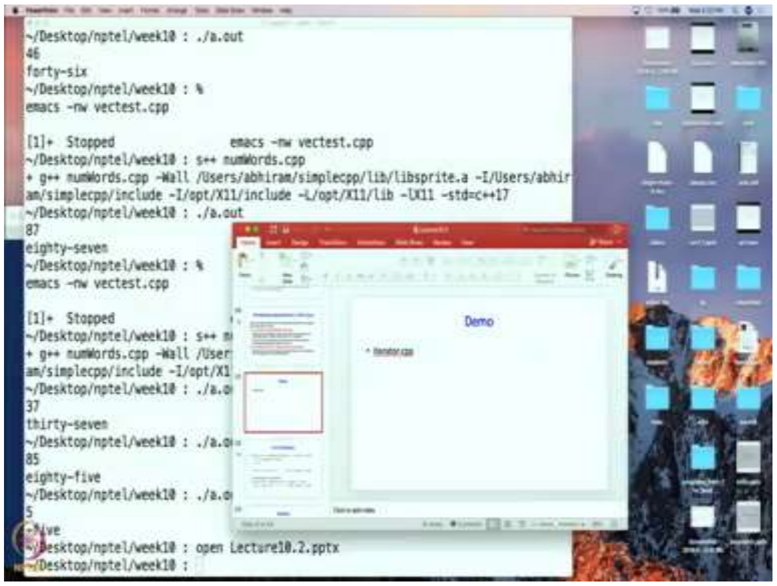
(Refer Slide Time: 8:10)

Demo

- iterator.cpp

So we will do a demo of this just to see what happens with ordered and unordered maps.

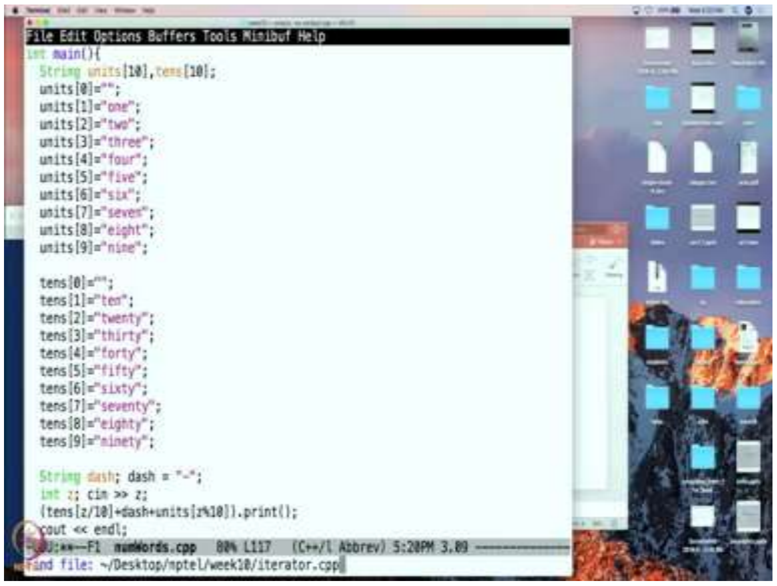
(Refer Slide Time: 8:19)



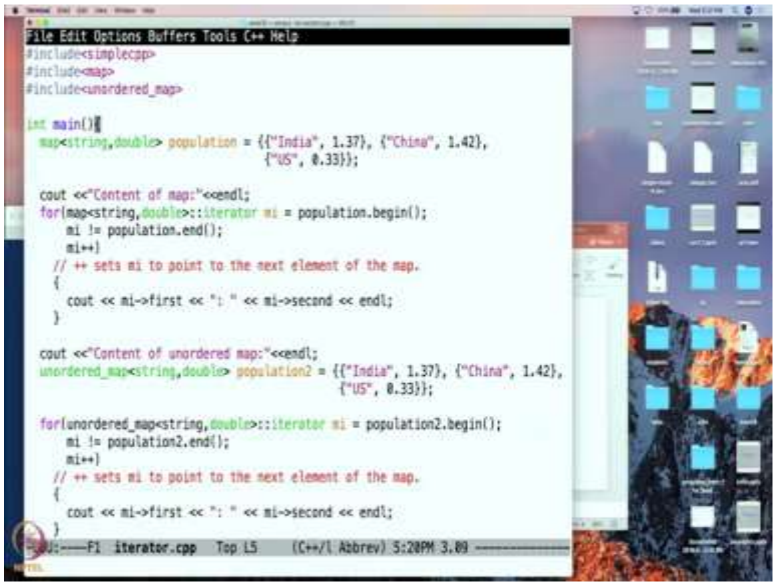
```
~/Desktop/nptel/week10 : ./a.out
46
forty-six
~/Desktop/nptel/week10 : %
emacs -nw vectest.cpp

[1]+  Stopped                 emacs -nw vectest.cpp
~/Desktop/nptel/week10 : s++ numWords.cpp
+ g++ numWords.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhir
am/simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week10 : ./a.out
87
eighty-seven
~/Desktop/nptel/week10 : %
emacs -nw vectest.cpp

[1]+  Stopped
~/Desktop/nptel/week10 : s++ n
+ g++ numWords.cpp -Wall /User
am/simplecpp/include -I/opt/X1
~/Desktop/nptel/week10 : ./a.o
37
thirty-seven
~/Desktop/nptel/week10 : ./a.o
85
eighty-five
~/Desktop/nptel/week10 : ./a.o
5
five
~/Desktop/nptel/week10 : open Lecture10.2.pptx
~/Desktop/nptel/week10 :
```

Demo

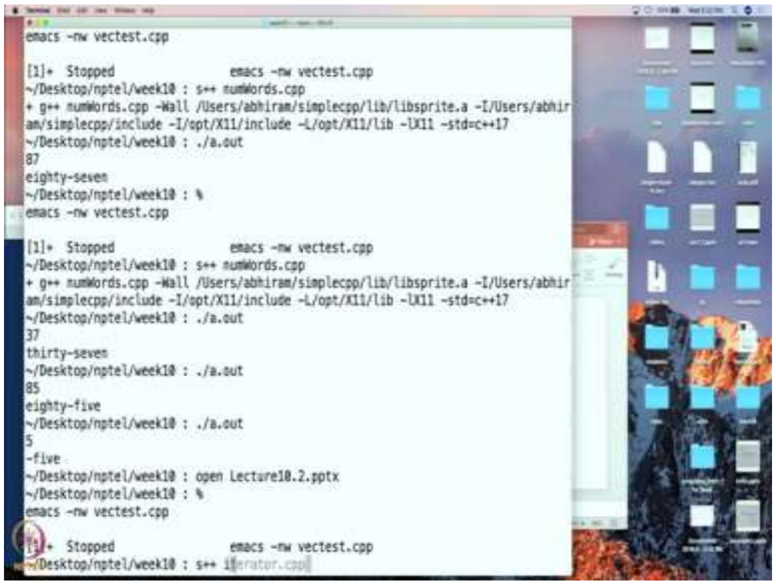
- iterator.cpp



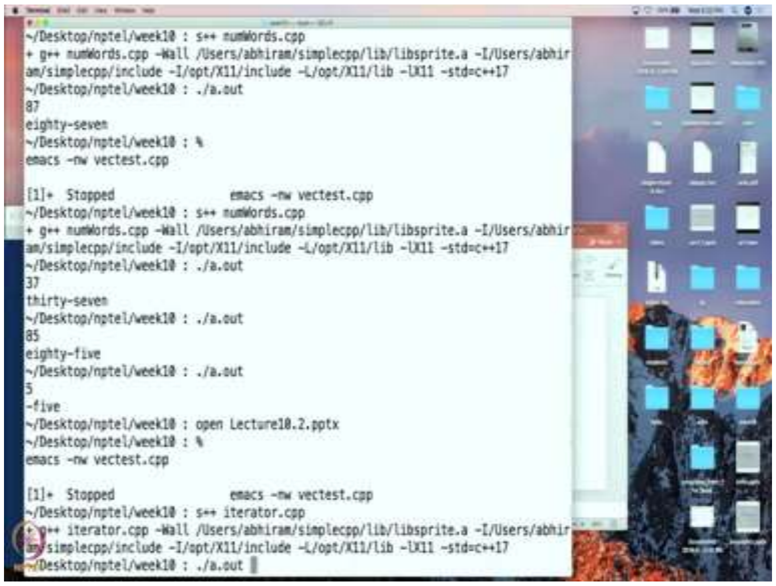
```
int main(){
  String units[10],tens[10];
  units[0]="";
  units[1]="one";
  units[2]="two";
  units[3]="three";
  units[4]="four";
  units[5]="five";
  units[6]="six";
  units[7]="seven";
  units[8]="eight";
  units[9]="nine";

  tens[0]="";
  tens[1]="ten";
  tens[2]="twenty";
  tens[3]="thirty";
  tens[4]="forty";
  tens[5]="fifty";
  tens[6]="sixty";
  tens[7]="seventy";
  tens[8]="eighty";
  tens[9]="ninety";

  String dash; dash = "-";
  int z; cin >> z;
  (tens[z/10]+dash+units[z%10]).print();
  cout << endl;
-U:**--F1  numWords.cpp   80% L117   (C++/l Abbrev) 5:20PM 3.09 ------------
Find file: ~/Desktop/nptel/week10/iterator.cpp
```



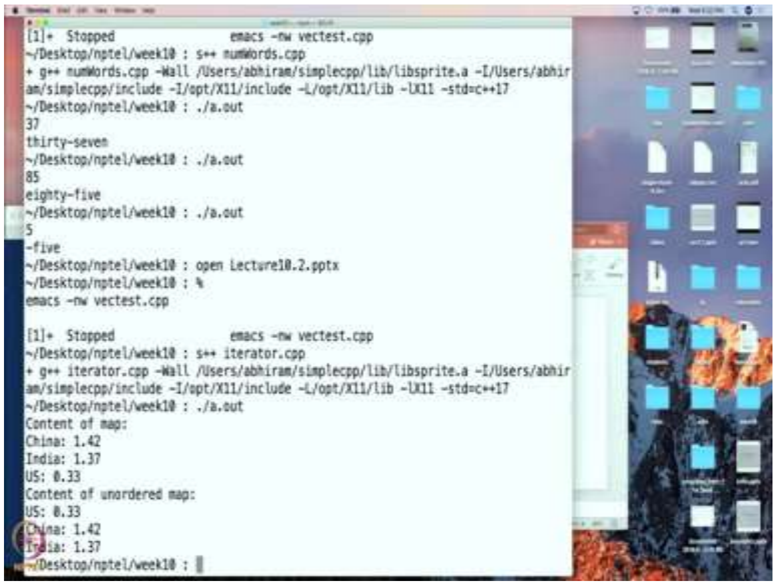
```
#include<simplecpp>
#include<map>
#include<unordered_map>

int main(){
  map<string,double> population = {{"India", 1.37}, {"China", 1.42},
                                   {"US", 0.33}};

  cout <<"Content of map:"<<endl;
  for(map<string,double>::iterator mi = population.begin();
      mi != population.end();
      mi++)
    // ++ sets mi to point to the next element of the map.
    {
      cout << mi->first << ": " << mi->second << endl;
    }

  cout <<"Content of unordered map:"<<endl;
  unordered_map<string,double> population2 = {{"India", 1.37}, {"China", 1.42},
                                              {"US", 0.33}};

  for(unordered_map<string,double>::iterator mi = population2.begin();
      mi != population2.end();
      mi++)
    // ++ sets mi to point to the next element of the map.
    {
      cout << mi->first << ": " << mi->second << endl;
    }
-U:---F1  iterator.cpp   Top L5    (C++/l Abbrev) 5:20PM 3.09 ------------
```



```
emacs -nw vectest.cpp

[1]+  Stopped                 emacs -nw vectest.cpp
~/Desktop/nptel/week10 : s++ numWords.cpp
+ g++ numWords.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhir
am/simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week10 : ./a.out
87
eighty-seven
~/Desktop/nptel/week10 : %
emacs -nw vectest.cpp

[1]+  Stopped                 emacs -nw vectest.cpp
~/Desktop/nptel/week10 : s++ numWords.cpp
+ g++ numWords.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhir
am/simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week10 : ./a.out
37
thirty-seven
~/Desktop/nptel/week10 : ./a.out
85
eighty-five
~/Desktop/nptel/week10 : ./a.out
5
-five
~/Desktop/nptel/week10 : open Lecture10.2.pptx
~/Desktop/nptel/week10 : %
emacs -nw vectest.cpp

[1]+  Stopped                 emacs -nw vectest.cpp
~/Desktop/nptel/week10 : s++ iterator.cpp
```



```
~/Desktop/nptel/week10 : s++ numWords.cpp
+ g++ numWords.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhir
am/simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week10 : ./a.out
87
eighty-seven
~/Desktop/nptel/week10 : %
emacs -nw vectest.cpp

[1]+  Stopped                 emacs -nw vectest.cpp
~/Desktop/nptel/week10 : s++ numWords.cpp
+ g++ numWords.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhir
am/simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week10 : ./a.out
37
thirty-seven
~/Desktop/nptel/week10 : ./a.out
85
eighty-five
~/Desktop/nptel/week10 : ./a.out
5
-five
~/Desktop/nptel/week10 : open Lecture10.2.pptx
~/Desktop/nptel/week10 : %
emacs -nw vectest.cpp

[1]+  Stopped                 emacs -nw vectest.cpp
~/Desktop/nptel/week10 : s++ iterator.cpp
+ g++ iterator.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhir
am/simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week10 : ./a.out
```



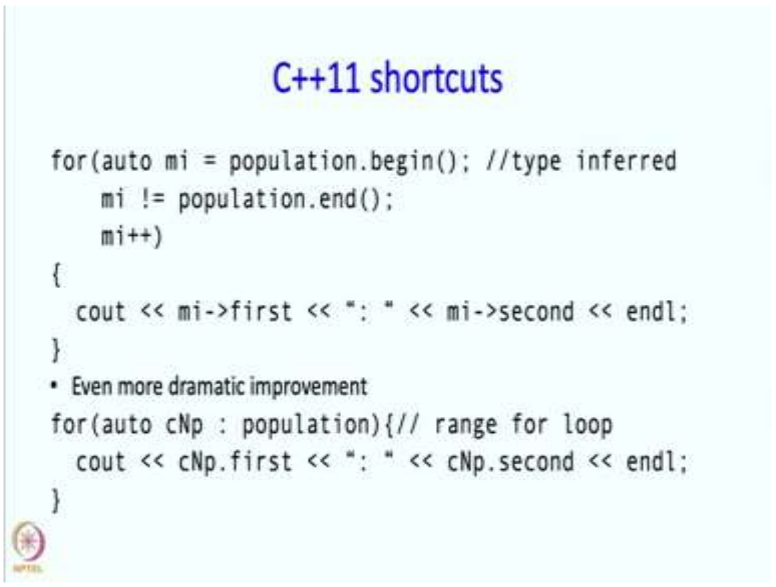
```
[1]+  Stopped                 emacs -nw vectest.cpp
~/Desktop/nptel/week10 : s++ numWords.cpp
+ g++ numWords.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhir
am/simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week10 : ./a.out
37
thirty-seven
~/Desktop/nptel/week10 : ./a.out
85
eighty-five
~/Desktop/nptel/week10 : ./a.out
5
-five
~/Desktop/nptel/week10 : open Lecture10.2.pptx
~/Desktop/nptel/week10 : %
emacs -nw vectest.cpp

[1]+  Stopped                 emacs -nw vectest.cpp
~/Desktop/nptel/week10 : s++ iterator.cpp
+ g++ iterator.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhir
am/simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week10 : ./a.out
Content of map:
China: 1.42
India: 1.37
US: 0.33
Content of unordered map:
US: 0.33
China: 1.42
India: 1.37
~/Desktop/nptel/week10 : 
```

So, let me get out of the presentation and, so let me open this file. So, here is our map. Oh, by the way I can initialize it in this manner as well. So, what does this mean. This is the, this is a pair that I want to put in. This is another pair that I want to put in. This is another pair that I want to put in. But the order over here is not the order in which the pairs actually go in. This is I can just put it in the order that I remember to pretend.

But C++ will store them in the right order. So, if you need the order it will be there. Now, what I am going to do over here is, I am going to do that going through all the elements but I am going to do this for, so I am going to do it twice. I am going to do this first for a map, and then later I am going to this same thing for an unordered map. So, for this map I am going to print out all these things. So, exactly the same thing, it is the same, it is the same loop going through all the elements in order. And here, well, I guess I should put this cout statement before I, after I define the map but does not really matter. I am putting in the same elements and I am going through, I am going through the same kind of loop. But this loop is for an unordered map. So, the iterator is for an unordered map.

And my original map is an unordered map. So, let us see what happens. So see what happened. So, for the map the contents were printed in alphabetical order or lexicographic order. For an unordered map, they seem to have been printed in some strange way, who knows what way they have been printed. But any way, so the look ups and indexing will work exactly the same for both.

(Refer Slide Time: 10:50)



C++11 shortcuts

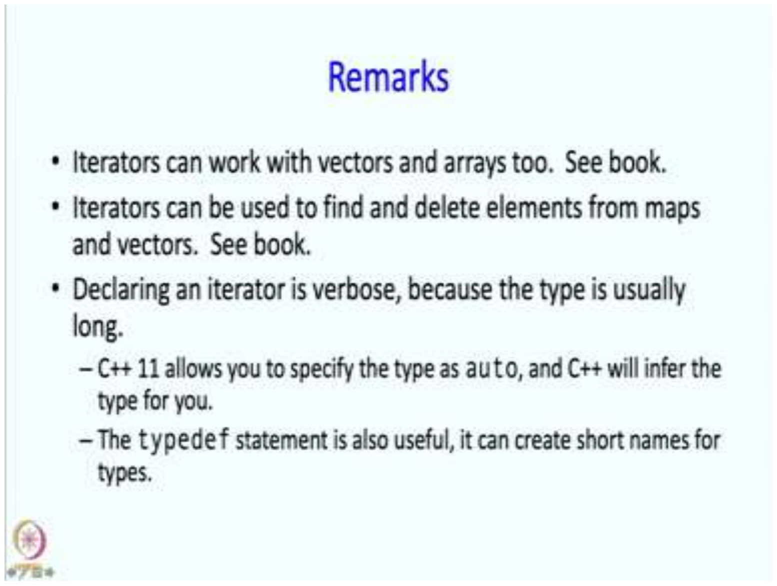
```
for(auto mi = population.begin(); //type inferred
    mi != population.end();
    mi++)
{
  cout << mi->first << ": " << mi->second << endl;
}
```

- Even more dramatic improvement

```
for(auto cNp : population){// range for loop
  cout << cNp.first << ": " << cNp.second << endl;
}
```

All right, so let us get back to our presentation. Yeah, so I want to talk about a few shortcuts which are of somewhat recent origin. So, instead of writing that whole complicated type expression, map::iterator, all of that, I can just put an auto over here. And then I can just say mi=population.begin(). So, this is an interesting statement because I am not actually giving the type, I am telling C++ that look I am assigning this to this. And so now you infer the type, so C++ knows that this has type all that map<….>::iterator. So, therefore, it will itself decide that this should be that type. So, saves us a lot of typing. The rest of the code is the same. But there is even more dramatic improvement possible. Another C++11 feature. So, this is what it is called a range for loop. So, here I am not even really using an iterator explicitly. I am saying after you iterate you will, so tell me, go directly to the objects that I am iterating through. So, do not even get me the pointers. Tell me what they are pointing to. So, cNp is really, are really the pairs the country, the country in population pairs which are stored in this iterator. So, this says that for all possible pairs, do whatever is in the body. And again since, this is a map not an unordered map, we will go over these pairs in increasing order of the index. So, I can just write over here cNp.first. By the way, this is not arrow because cNp is the pair itself. Not a pointer to the pair and here again cNp.second, that is it. So, this really makes it much nicer to deal with maps because all that, all those huge iterator references are not there at all.

(Refer Slide Time: 13:25)

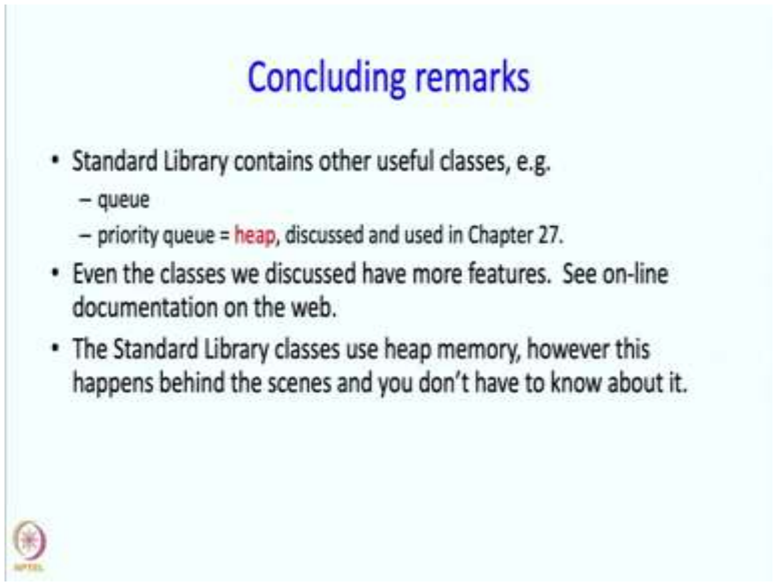

Remarks

- Iterators can work with vectors and arrays too. See book.
- Iterators can be used to find and delete elements from maps and vectors. See book.
- Declaring an iterator is verbose, because the type is usually long.
  - C++ 11 allows you to specify the type as `auto`, and C++ will infer the type for you.
  - The `typedef` statement is also useful, it can create short names for types.

So, iterators can work with vectors and arrays. So, far that please see the book. Iterators can be used to find and delete elements from maps and vectors. Declaring an iterator is verbose, because the type is usually long. So, as we just saw you can specify the type as auto and C++ will infer the type. And occasionally it might be useful to define to use a typedef statement.

This is also defined in the book. So, in fact, instead of writing say something like vector of vectors, vector of vector of end. I can call it a matrix. So, wherever I want to type vector of vector of end I just need to see matrix. So, typedef statement allows me to do just that, again for the syntax of it, please look at the book.

(Refer Slide Time: 14:23)



Concluding remarks

- Standard Library contains other useful classes, e.g.
  - queue
  - priority queue = heap, discussed and used in Chapter 27.
- Even the classes we discussed have more features. See on-line documentation on the web.
- The Standard Library classes use heap memory, however this happens behind the scenes and you don't have to know about it.

All right! So, that brings us to the end of this lecture. So, I just want to make a few conclusions. So, standard, a few remarks, the standard library contains some classes which we discussed but it also contains some other classes. So, there is a queue class. There is a priority queue class which is discussed in the book in Chapter 27 and several other classes. And even the classes that we have discussed have a lot more features. For that, see the online documentation on the web. The standard library classes use heap memory. However, this happens behind the scenes and you really do not even need to know about it. And that is the beauty of it. It is reducing your thought, your thought overhead. You can think of these as just as abstract mathematical objects. So, a map there is a mathematical object called the map. So, the C++ object called the map is very-very similar to the mathematical object called the map. So, it is a nicer way of thinking about things rather than saying that oh there is memory allocation happening. There is, it is happening but you do not have to think about it. These classes are extremely useful. So, definitely, get lots of practice with them. And solve the problems at the end of the chapter and we conclude this lecture over here. Thank you very much.