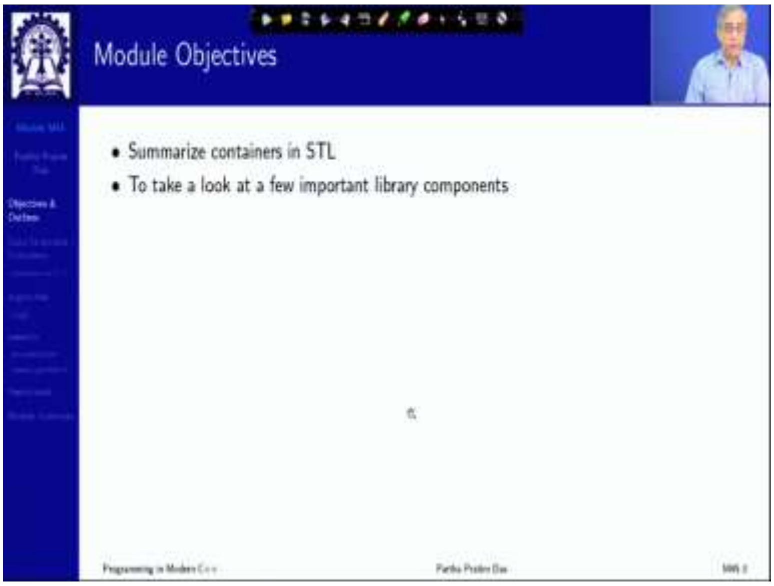
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 45
C++ Standard Library: Part 3 (STL)

Welcome to Programming in Modern C++. We are in week 9 and I am going to discuss Module 45.

(Refer Slide Time: 00:37)

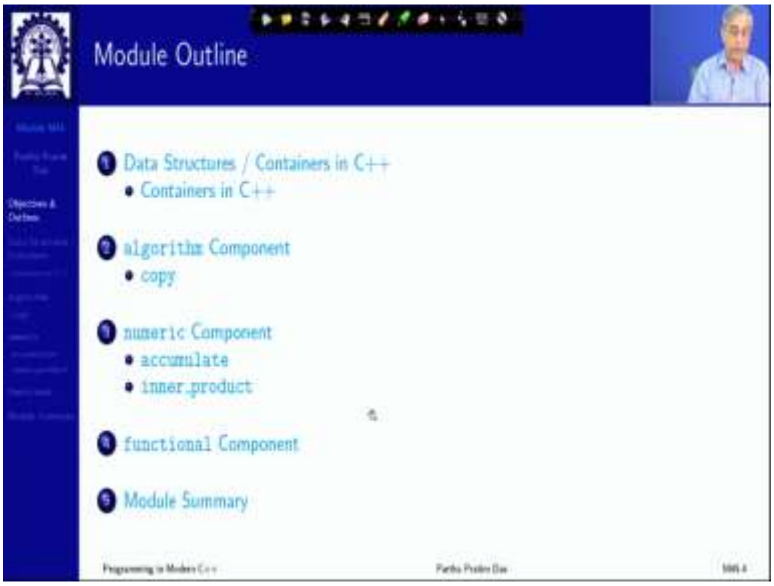
In the last two modules, we have been discussing about C++ standard library. Specifically, we took a look at generic programming. And in the last module we discussed about certain common properties of STL and its use in terms of the containers, useful containers that we have.

(Refer Slide Time: 01:01)



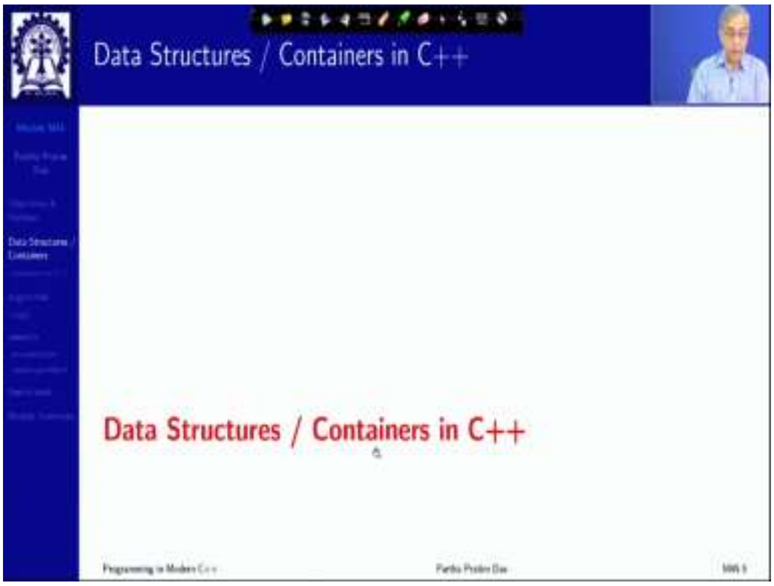
In the present module, we will summarize the containers in STL. Certainly, we are not going to discuss each container at a depth as we did for vector or map, but we will summarize and show you the commonality between them. And we also take a look at few important other library components, we are associated for the use with the containers and even otherwise.

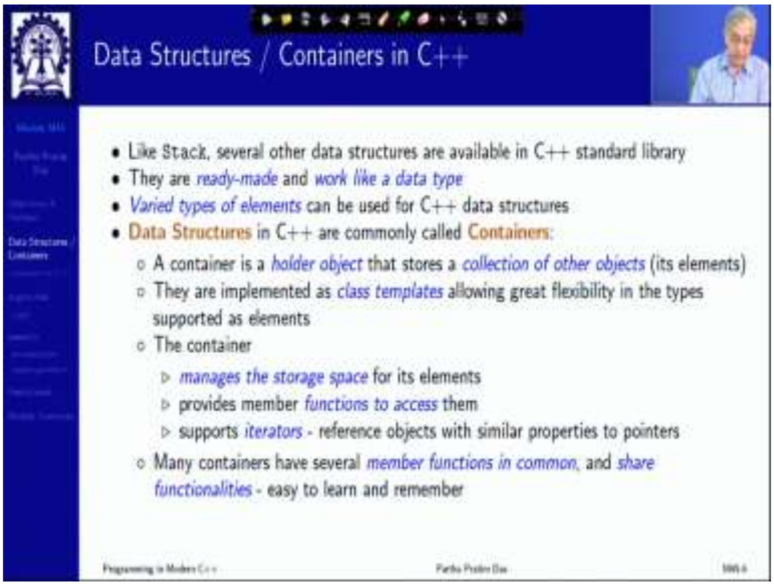
(Refer Slide Time: 01:29)



Here is the outline which will be available on left.

(Refer Slide Time: 01:35)



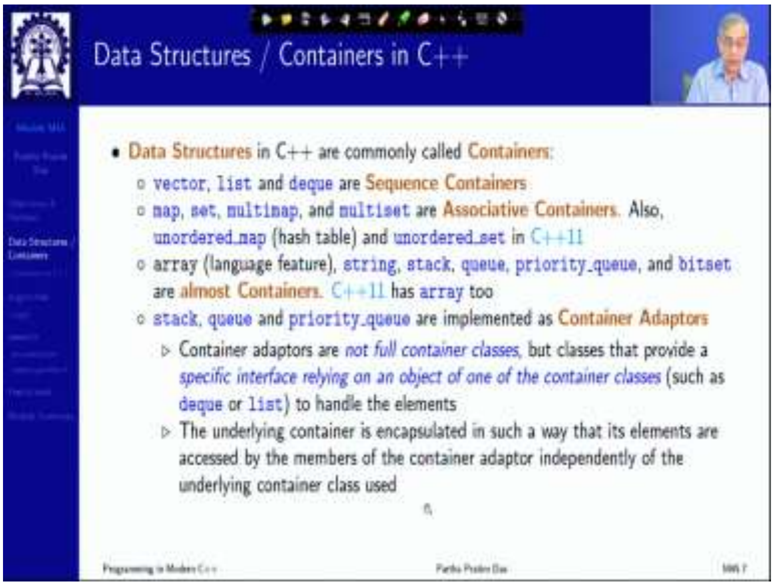


So, let us try to take an overall view on the data structures or containers in C++. So, containers are data structures in C++ standard library. They are readymade and they will work completely as a data type. That is a very very important thing that they are not just data structures as in C but they are data types.

So, anywhere you can use a built in data type, you can use a container. So, that is the kind of parallel that happens and the varied element types can be really varied, including of course, user-defined types, built in types and so on. So, a container is a holder of object that stores the collection of other objects, depending on the underlying type.

And they are typically implemented as class templates which allow the great flexibility of types that are supported as element. It manages the storage space, it provides member functions to access and it supports iterators. Now, you know that supporting iterator is very very important to use the container because that is the only way you can write algorithms for the different containers, ok.

(Refer Slide Time: 03:03)



Now, there are containers are kind of classified in certain sub classified in certain ways. vector, list and deque are known as sequence containers. vector you know, list is doubly linked list. deque is double-ended queue; typically, I mean many people pronounce it as deck. So, as in a queue, you can add at one end and remove element from the other, in stack you add and remove elements from the same end.

In deque you can add and remove elements at both ends, so that is that is why it is called doubly ended queue. But they keep the elements in a physical sequence. So, they are called sequence containers. Then we have associative containers associative containers are those like a map, where you have a key and a value associated with it. So, here in a sequence container, you find out the value based on certain position in the sequence, either through indexing or by traversing the list or by taking adding and taking out elements from deque and so on.

But in associative container, there is an association between a pair of values. So, given one you find the other, so that is why it is called the associative container. And map is the most useful associative container or the most useful container after vector which can associate any key type with any value type.

Set is another which is a collection of value items which are unique. So, it just allows you to have unique set of elements which is very important and you can do typical set algebra with that. multimap and multiset are relaxations on the map and set where you have allowed duplication of elements, like map will not allow duplication of key, but if you want duplicate keys with different values associated with them to be present then you can use a multi map.

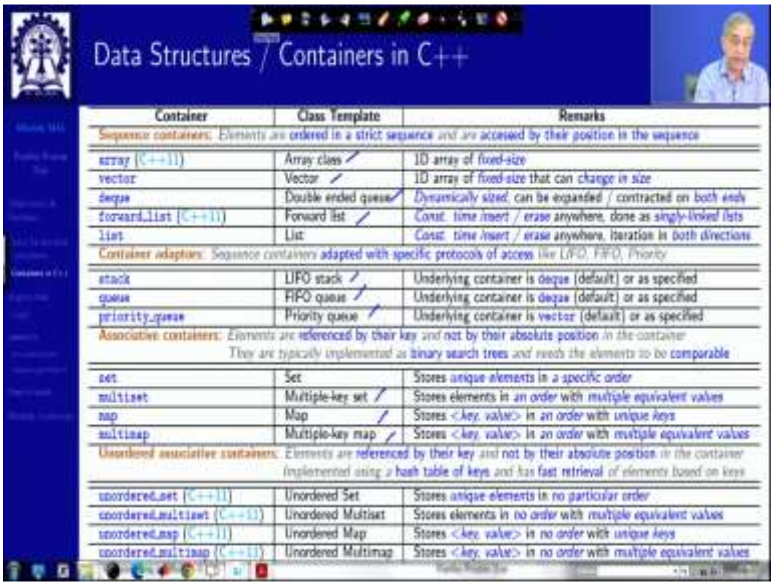
Similarly, you can use a multiset. In C++ we will see there is also things like or unordered map which is basically hash table or unordered set because even though we do not say in terms of the associative container, the underlying implementation of these containers do need an ordering. And since they do need an ordering, the element type should be such that the ordering should be possible, they should be comparable.

But in unordered cases you would not require that. And many are called almost containers. So, primary of them are container adapters which are not total implementation of containers but those are implemented on other containers with certain additional property. So, you have stack in that, you have queue in that and you have priority queue in that.

These are container adapters. So, they have an underlying container which is not necessarily always specified. And based on that, so that container gives you the basic container support. But there are specific functionality that you implement in terms of the member functions in the STL that gives you the stack component, queue component and priority queue component. So, these are called almost containers, so is string. Because it is kind of a vector but its element type is always character. So, it is a kind of almost container.

General arrays that we have in the language is almost container. bitset is where we keep the bits is an almost container and so on. And in C++ 11, we will see that the standard library will also have a component called array other than the language array. And that array is different from the vector, we will see that one.

(Refer Slide Time: 7:29)

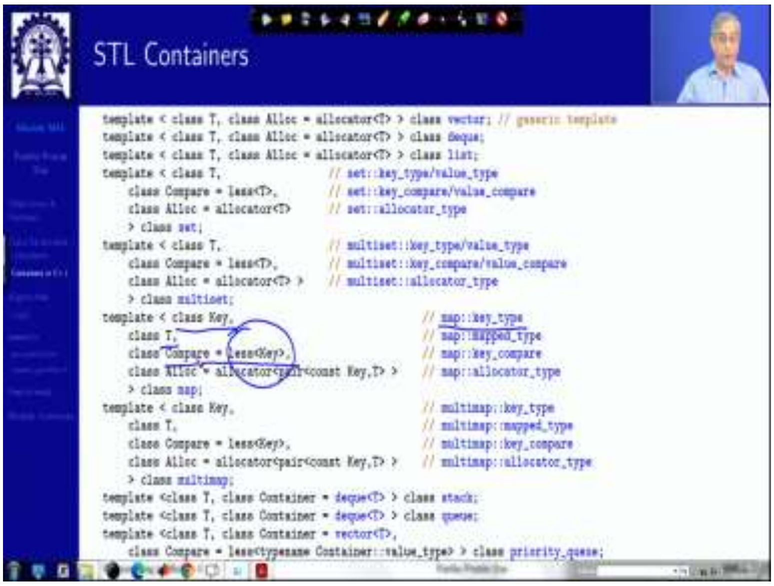


So, this is a complete chart of the containers in the standard library. Just for convenience and easy reference you remember my one slide summary principle. So, this is a one slide summary of the containers that you have in C++ including C++ 11. So, those which are only in C++ 11 I have marked them.

And what you can get to see is what is the class template for the each one. So, which basically say what is the functionality that this particular container will have. And here are some remarks that are available, like for stack the underlying container by default is a deque, whereas for a queue it is also a deque, whereas for a priority queue it is a vector. But the design is such that, if you want then any other type of underlying container, as specified, can also be used for these container adapters.

Now, there are different properties, basic properties that I have tried to summarize here. So, you have the sequence containers, three in C++ 03, two more in C++ 11, three container adapters, four associative containers and four associative unordered containers in C++ 11. So, this is the total set. Of course, you do not; from day one you do not start using all of them. It is primarily the vector and map, then possibly list, stack, queue is what will be most of the use that you will find.

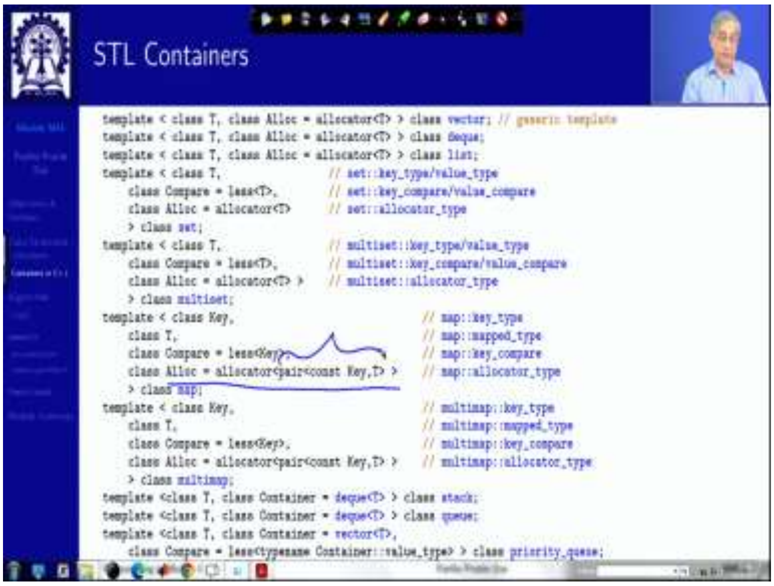
(Refer Slide Time: 09:30)



These are just showing you the template styles of these different containers in C++ 03. I am not going through them, you can read them, study them. And the basic principle you have to remember is of uniformity. So, do not try to remember anything but try to understand the reason of why it is.

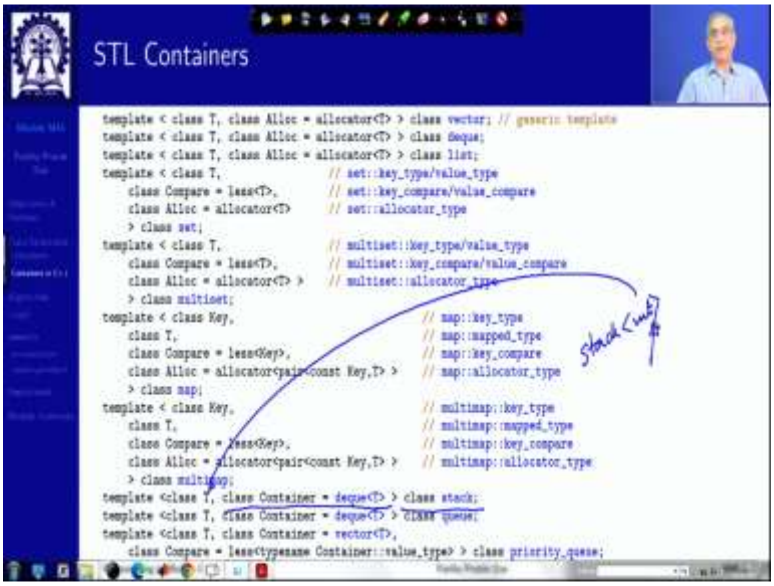
So, for example, if I look at say, map I am looking at map. So, class key obviously is the key, this is T is a map type and I have a compare. Why? Because as I said, map is an ordered container. To represent the map in the underlying way, it is using a binary search tree which needs ordering. So, I need to have an ordering on the key value. So, I am using the Less functor with the key value type for doing this.

(Refer Slide Time: 10:34)



What will be the allocation? Allocation is allocator is basically the underlying container that you use. So, this is the type of the allocator that you will have. It will have a key, constant type of key and the map type paired and allocation will happen on that. So, once you understand this, you will you will understand that it is relatively easy. You do not have to really remember anything but things are done in a very very uniform manner.

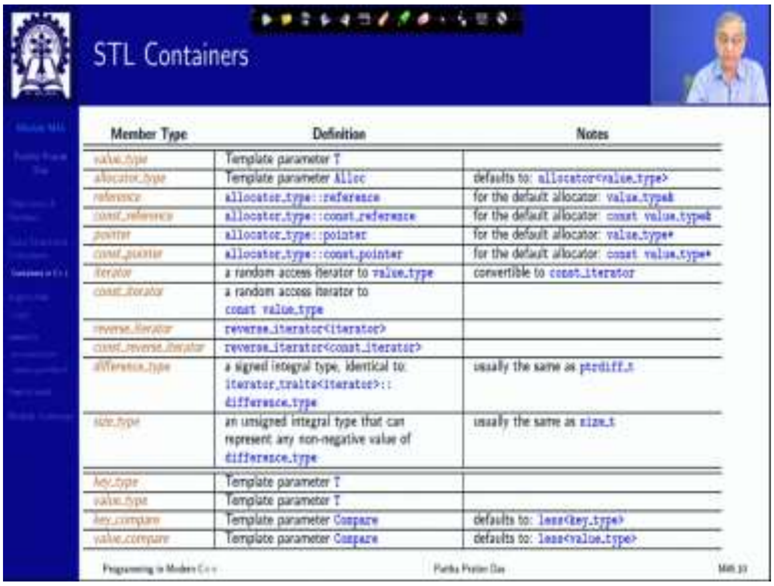
(Refer Slide Time: 11:10)



For example, in case of container adapter stack, it is saying that the class container, the second parameter is class container, which is defaulted by deque T. So, if you just say stack int, if you just say stack int whatever you specified you have specified this T as int. So, your stack will actually be implemented, you will get a code that is implemented on deque<int>, deck of integers.

But if you want something else, you can pass a second parameter to your template instantiation, of what type of underlying container you want and you will get that type of container. So, that is the kind of flexibility that STL containers give us. That is the kind of uniformity STL containers give us. There is a kind of power STL containers give us.

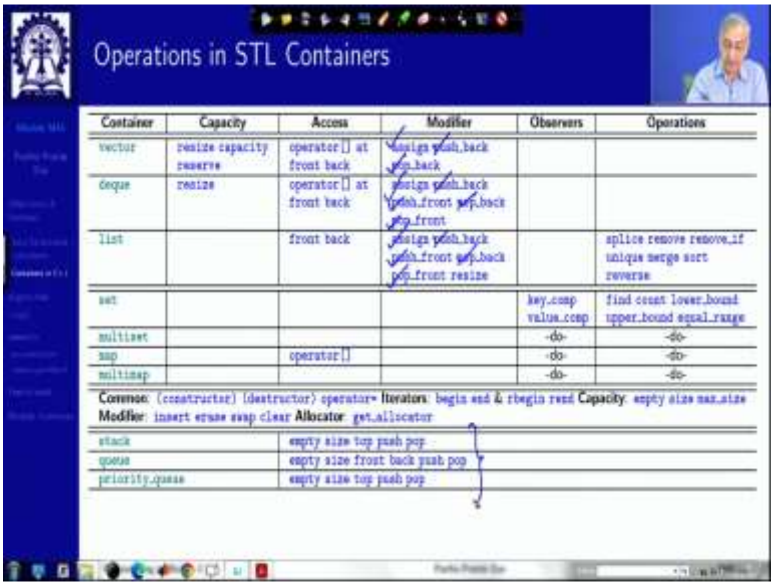
(Refer Slide Time: 12:21)



So, these are the different member types. value_type you have seen already number of places. allocator_type is what kind of allocation will happen. Then there is reference_type, pointer_type and so on. iterator_type we have seen, key_type, size_type. So, this is typically what. May be some containers will not need to define some of these but most containers will define most of these member types.

So, you know conceptually, that over all these 10 containers, it is all uniform in terms of. So, these are the types that you can use the container type name vector<int>::value_type or, so, you have that type you will be able to see what is the value_type that you have, what is the size_type that you have.

(Refer Slide Time: 13:16)



Coming to the operations, I have tried to do a uniformity summary. I was looking for it; I did not find it anywhere. So, I built it from the manual. So, on left you see the containers and these columns are the types of different operations that are available. So, one is capacity related.

So, vector has three capacity related members resize, capacity and reserve. Whereas deque has only one. List or set etcetera, they do not have anything. Whereas, if you look at say modifiers, you will find this as a push_back, pop_back that is you can add at the end, take out the, this is push_back, pop_back, but deque is at two ends. So, it has push_front, pop_front.

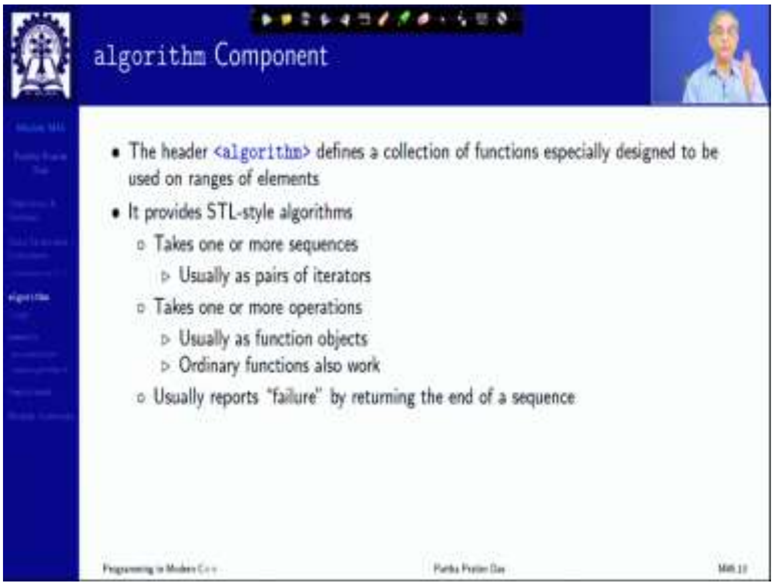
List is at both ends. So, it has all of these. So, assign is available for all of them. You can assign each one of these data structures. So, you can see that there are different, if you just think conceptually as to what the data structure should give you, you will find that those operations would be available appropriately in that STL container type.

So, here that is some more of these, some we have already seen. And like if you look at the container adapters, you can see a very very uniform design. All have empty, size. Stack has push, pop. Queue also has push, pop. Priority queue also has push, pop. In addition, to keep to the naming conventions used in queue, it also has front and back. So, these are different kind of operations that you can think of.

(Refer Slide Time: 15:20)

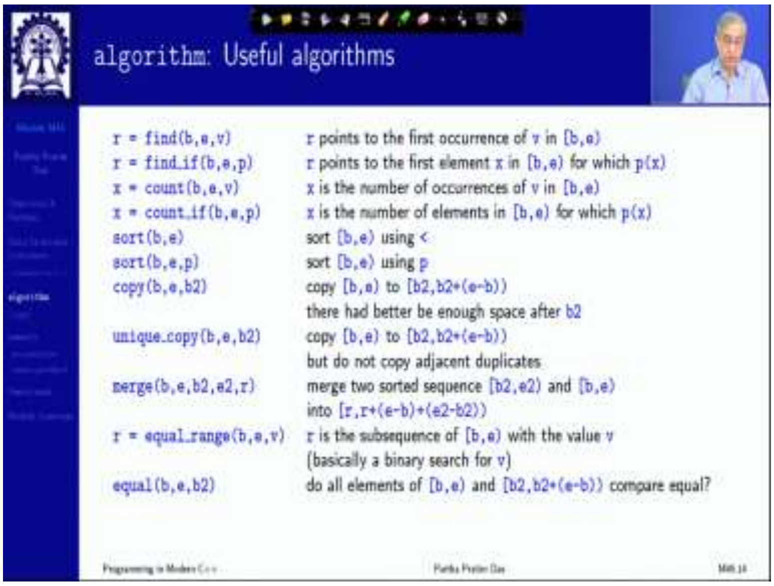
So, kind of that was a summary picture of what you have in terms of containers in STL. Before we close on STL and the standard library, let me just take you through some of the other components like algorithm.

(Refer Slide Time: 15:31)



Algorithm is a component a header which defines a collection of functions which are designed to work on a range of elements. Where do you get a range of elements? By iteration. So, that that is how algorithms can work on any container. So, it is naturally STL style. So, it takes one or more sequences and one or more operations and performs.

(Refer Slide Time: 16:05)

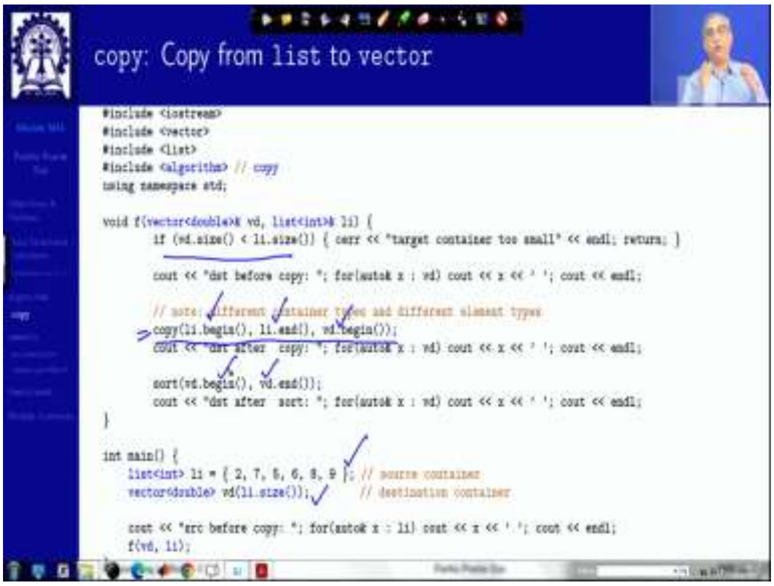


Naturally there is a large number of algorithms available but some are like find, is b, e, v written in short form so that I should a lower end notation also I can give that. But it is quite obvious, b is the beginning iterator, e is the end iterator, v is the value. So, you are doing a find you have already seen that. Find b begin, end predicate. Similarly, count; it counts how many are there of that value v and so on. So, find, count, sort, copy, unique copy, merge, these are some of the very common algorithms which will be available in this component.

(Refer Slide Time: 16:51)

Just to see an example say of copy. So, we can copy the elements from a range of first to last, to another iterator starting at say result. So, it has to take two iterators, one is the input iterator from which you are copying, and another is the output iterator to which it goes. So, the output iterator finally is a result that you return. So, in the input iterator you get the first and last, output iterator you just have the result where you will copy, the destination. Quite simple as to how the code will look like.

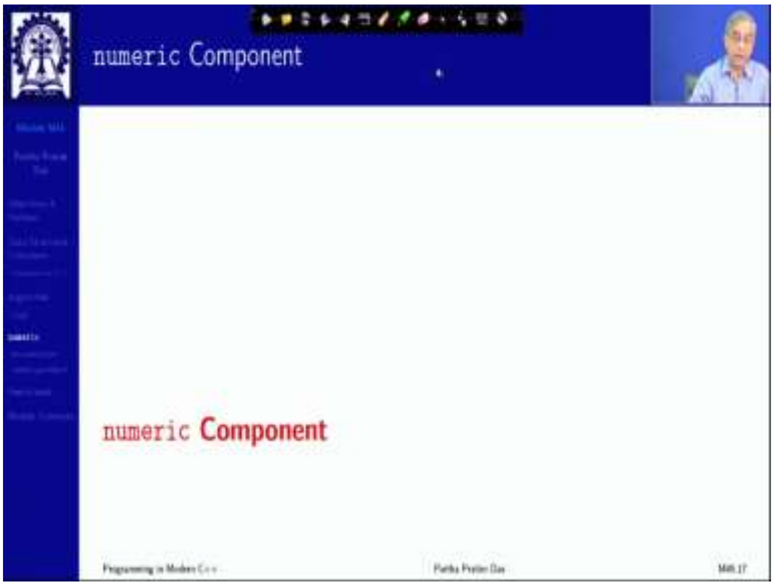
(Refer Slide Time: 17:32)

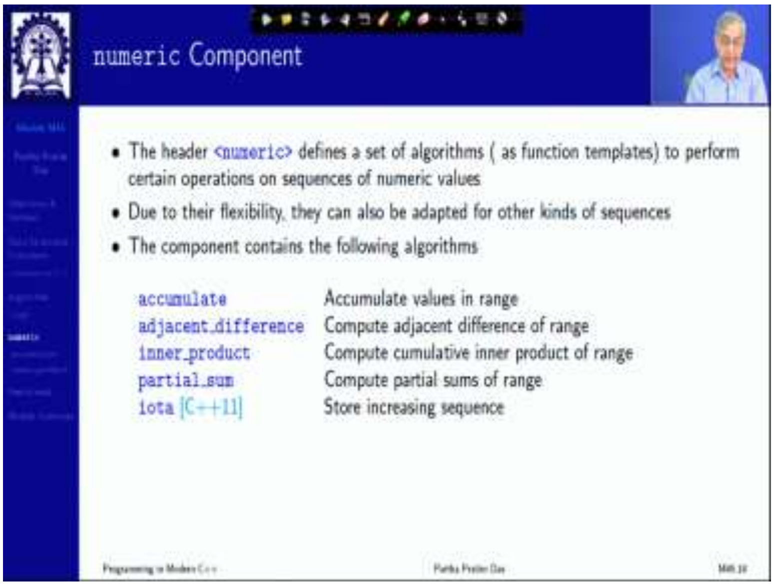


It is you do not need to write this because it is available in the algorithm component. Just include that you will have a copy. So, here I have defined a list of int and a vector of double. And I have made the size of this vector same as that of the list because I want to copy in that. So, I am copying elements from a list of integer to a vector of double. And just see how easy it is. First I will check if the vector is large enough, if it is not then, certainly, I cannot do.

And then copy is just a one-line code. This is the beginning iterator of list, ending iterator of the list. So, this is my list. And this is the beginning iterator of my destination output iterator, the vector, where I will copy. So, once I have done that, it is all available. I can assign this to get the final iterator if I want to use it. I have directly used the vector itself and I have sorted it using the sort function which is also available in the algorithm. So, once I do that naturally, you can see that how easily things can be copied and sorted using the algorithm component.

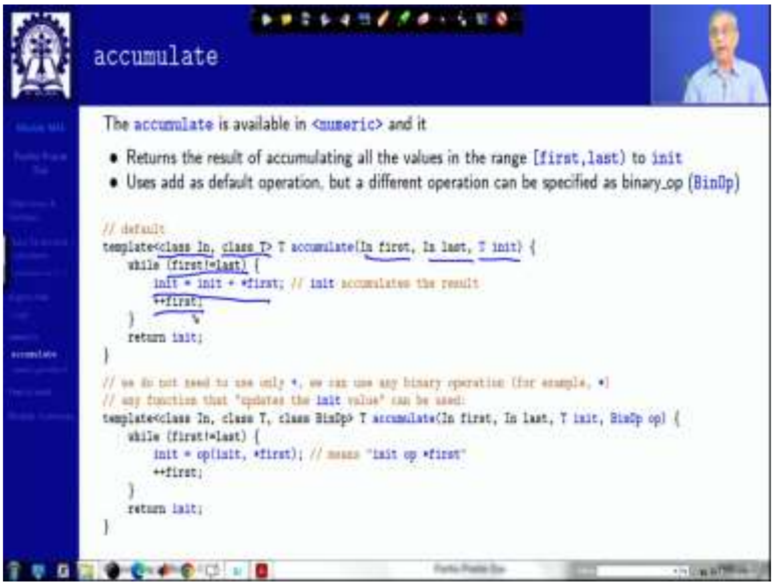
(Refer Slide Time: 19:04)





There are very interesting numeric component available which can also be used in semi-numerical context. So, the numeric header, basically, had in C++ 03, it had 4 different algorithms. In C++ 11, one more has been added. So, of these I will just take example of two and show you.

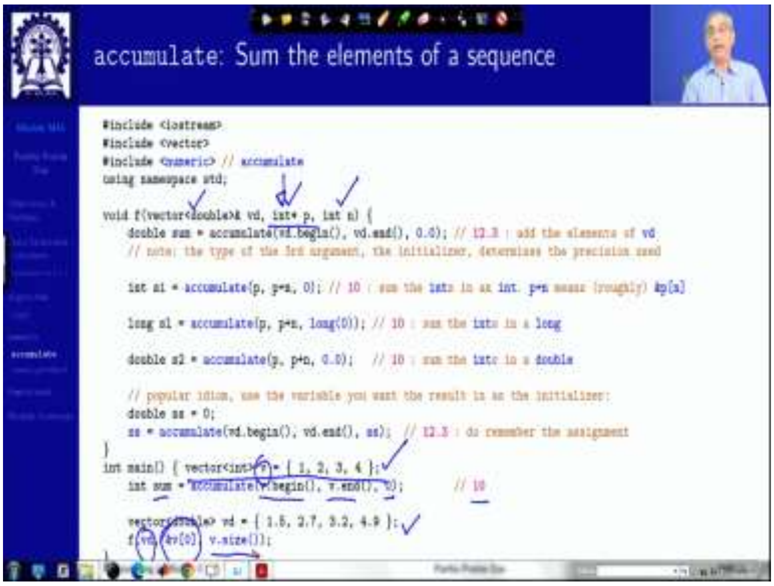
(Refer Slide Time: 19:31)



First is accumulate. I really like this because what it does is, it basically you can think of that I have a collection, I want to take the elements and add them. So, it simply does that accumulate. So, the iterator class as input, the element type to add. So, your actual function is the first and last iterator and the initial value. Because you need to, you are accumulating, so are you starting with 0 or starting with something else, so that tells you the element type.

And this at least, this is your iteration loop which is by now you know, it is almost common across all different algorithms. This is the way to go to the next element, this also is known. So, this is my accumulation code. You take the element *first, add it to end, put it back to end. So, that is a beautiful logic.

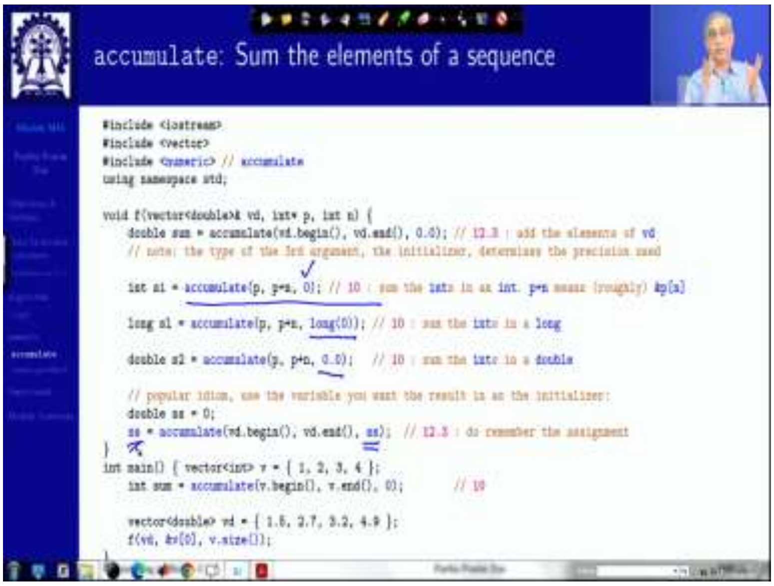
(Refer Slide Time: 20:43)



Now, before getting into the next one, let me just show you the use of this. So, this accumulator, with this accumulator I have created a vector and I accumulate, begin, end. I have passed 0 as the initial value. So, starting from 0 all will be get added. 1, 2, 3, 4. Result is, if you print sum, you will see 10.

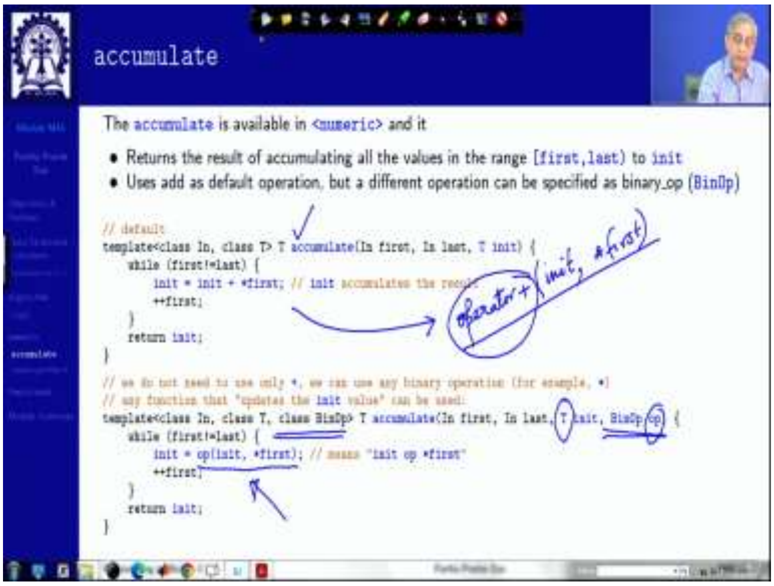
I have another vector of double and I have written a function which takes a vector of double. It takes an int pointer and it takes a number. So, just to show the different ways of working, so in the vector of double, I pass vd, in the int pointer I pass this vector by the address of its 0’th element. This address of it is 0’th element. I pass it here. So, it is coming as an integer pointer converted and then how many elements are there in that vector.

(Refer Slide Time: 21:50)



So, as I do that inside this, I can accumulate. If I pass 0, it will accumulate in the integer way. If I pass long zero, it will accumulate in the long type. If I pass 0.0 it will accumulate in the double type and so on. So, all these are possible. I can also pass my accumulation variable and then reassign it here to get the accumulation done. So, you can accumulate everything. That is a nice way that you do not need.

(Refer Slide Time: 22:29)

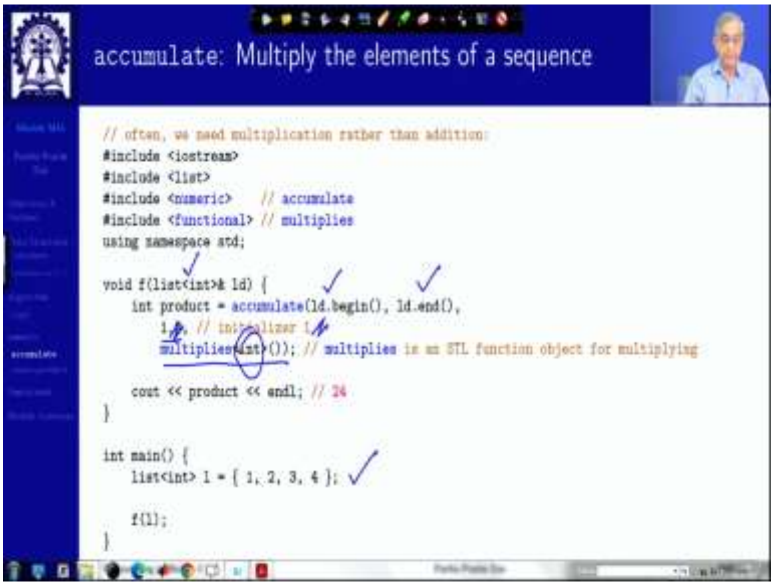


But, the more interesting part of this story is that you can actually, I have actually come back one slide. You can, if you notice here then this is just an operation of addition being done. It is an operator. So, with all our notion, I can say that this is nothing but operator + working with init and *first.

So, it is possible that I will not hard code this operator +. Rather, I will pass it to this algorithm as a function object. So, I put another parameter, binary operator. I put another parameter to this accumulate function also. And I pass an op which is a binary function object which certainly has to return an appropriate type of value which is same as T.

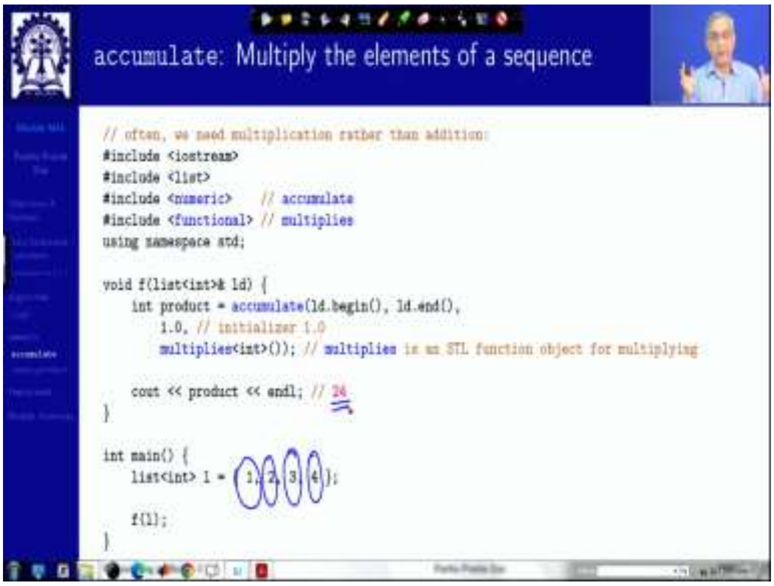
So, then this generically is changed to op applied onto in init and *first and that will do the job. Now, if I pass operator +, of course, I will not, because I have the default accumulator. If I pass operator + then I will get the first behavior, otherwise, I can pass anything. I can pass an operator multiply. Then the values will get multiplied as nice as that the same code.

(Refer Slide Time: 24:21)



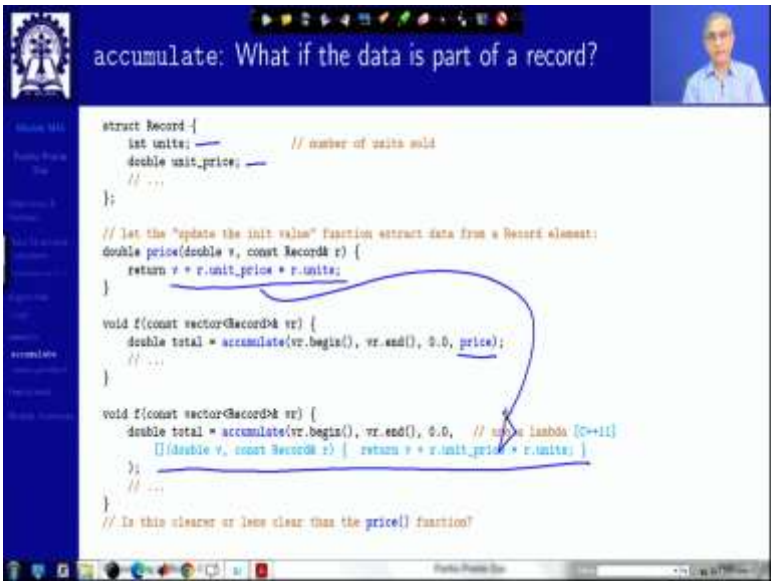
So, let us go and see this. So, I have this accumulate. This is a list of list l of integer 1, 2, 3, 4. I have defined and then I am calling this function with this list. I have begin and end iterator. I am initializing with, actually I should initialize it with 1. 1.0 is not needed. Initialize it with 1. And then what I pass as an operator is multiplies. Multiplies is a standard function object which is a binary function object which takes two parameters of the given type, multiplies them and returns back the result. So, it is a binary multiplying operator. So, I pass the multiplies. So, what will happen?

(Refer Slide Time: 25:22)



Now, instead of adding the elements, as they come from the iteration, it will multiply the elements. So, it will multiply 1 with 2 then with 3 then with 4. So, the result will be 24. So, these are, and I mean this is just a simple illustration. You can write your own functor to pass here as well and do anything as you accumulate the values.

(Refer Slide Time: 25:52)

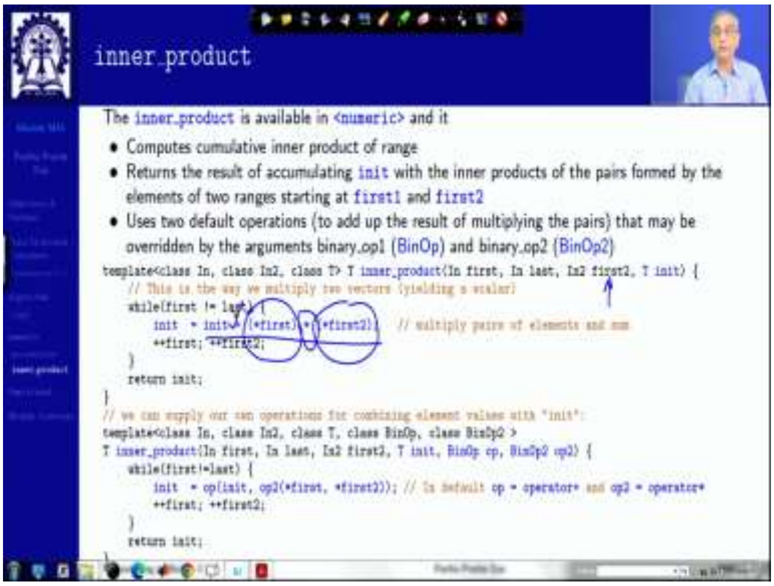


The accumulation can be done in terms of components of records also. Suppose you have a record which has unit price and number of units and you want to see what is the total price, total value of the collection. So, for every case, every record in that collection, you have to multiply the unit price with the number of units and add them.

So, it is not a matter of one operation but it is a multiplication then addition. So, you can define it in terms of a function and pass that function pointer. Just remember, anywhere you can pass a function object; you can also pass a function pointer. Or better still you can define a lambda which is a function object.

There is an anonymous function with the same code as this one and pass it to the accumulate. It will accumulate the entire value. The advantage of doing this is, of course, since this is a simple code, the advantage of doing this is whoever is reading it can clearly see what is being done. Very understandable. An anonymous function, we will see more in C++ 11.

(Refer Slide Time: 27:22)

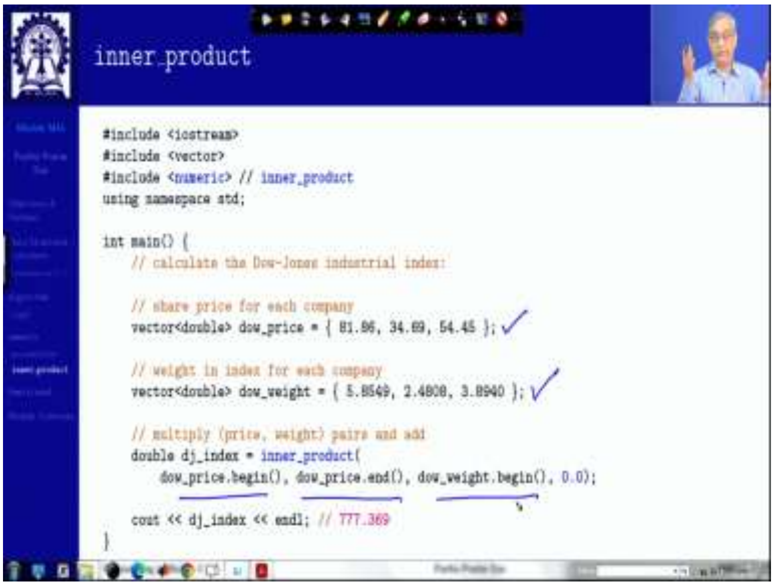


So, these are some of the different things that accumulate can do. Let us look at another inner product. Inner product is something in school, high school you used to do in a vector inner product. That is you are given two vectors and you do component wise multiplication and then add them.

But now for us, it need not be a vector because everything through the iterator can be thought of as a sequence. So, it is a corresponding elements of the sequence, those will have to be multiplied and added together. So, an inner product component, inner product will, algorithm will take an iterator, first and last of one sequence, one iterator in.

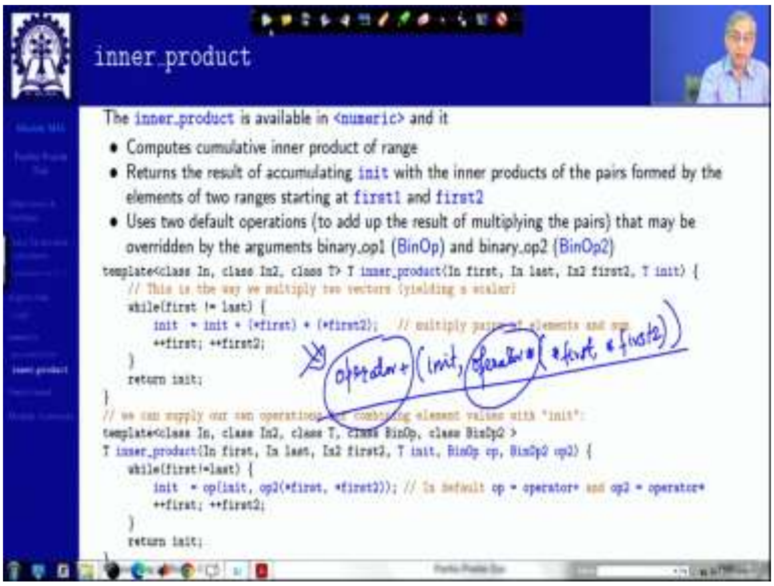
Then it will have another, the other one, the other vector or other list that you are doing inner product with. You do not need the end of that because when this ends that has to end because unless they are of the same length inner product is not defined. And the initial value. So, kind of the price, unit price, price part we were doing, can be directly done by this. So, you get the element pointed to by the first iterator, get the element pointed to by the second iterator at two different vectors, multiply them here and add accumulate. So, this gives you the inner product.

(Refer Slide Time: 29:03)



So, you can see. There is a very naive example I have given. These are different prices and there is a different weights of the index and so you dow_price.begin, price.end and weight.begin and find the inner product to see the entire value.

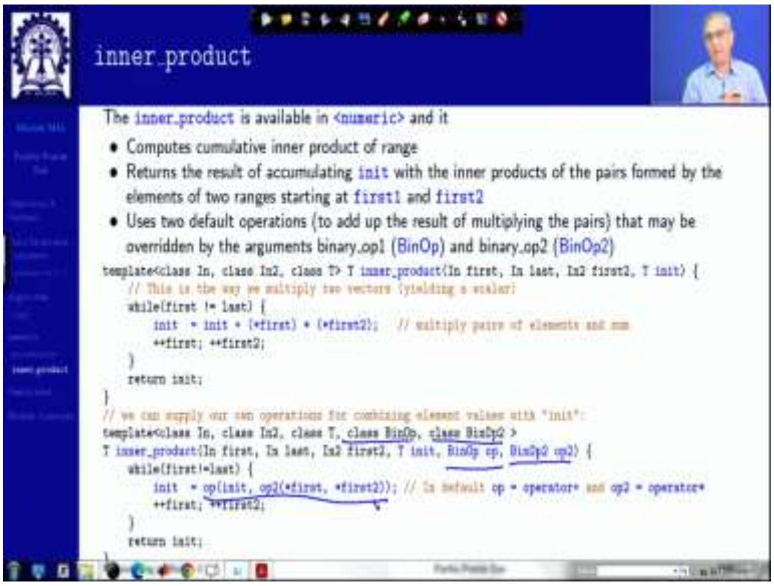
(Refer Slide Time: 29:31)



Interestingly, like you do for, like we did for accumulator, we can generalize the inner product also. But in inner product, there are two operations. What you are saying? You have sequences; you are taking element component wise, each l element, multiplying them and then adding them.

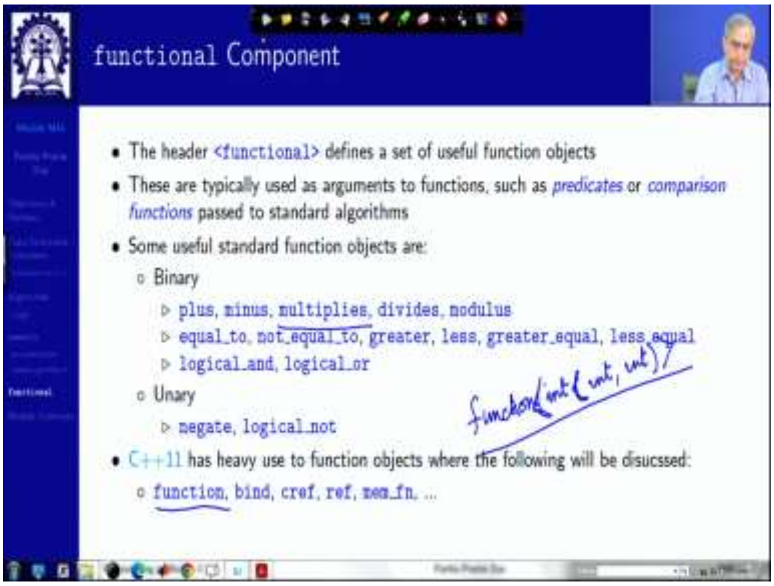
So, this is operator +, init, init then the becoming cluttered, let me write clearly. Operator + then init, then operator *, then *first, then *first2. This is the meaning of this expression. So, there is nothing special about operator + and operator *. I can pass two functors.

(Refer Slide Time: 30:38)



So, the generic form of this actually passes two binary operators, two binary function objects and does this as a total generic binary operators of any two types. So, you can do inner products. For example, I can define, interestingly, I can define that I will take element wise add them and then added value I will multiply, whatever that means. So, anything of that sort is available. So, you can see the power of STL. It gives you; it is a lot of flexibility and generality of what you can do with the data with the containers.

(Refer Slide Time: 31:31)



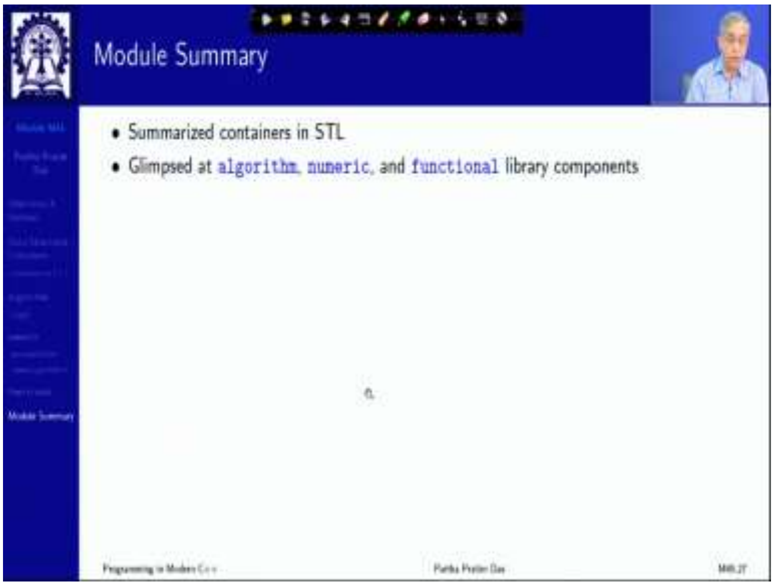
The functional component is another very important which, basically, defines a set of function objects. So, these are used typically as arguments to functions such as predicates or comparison functions and so on. So, there are many useful function, standard function objects like plus, minus, multiplies, we have just now we have used, multiplies, divides, modulus.

So, the basic arithmetic operators, the basic your comparison operators, logical operators, those are available in terms of the functional component. So, when you are making use of STL algorithms, you can make use of these function objects to ease your task. You do not have to write them. But a much bigger value of the functional component comes in terms of actually building up bigger components, more powerful components in terms of building up closure objects and so on so forth.

So, there are some like, there is a template called function which allows you to define the prototype of a function having certain input parameter types and a given result type. So, you can write something like function int. I am sorry. This is function. This is you can write something like this. Which will mean that this is the type of a function which takes two integers and gives you an integer result.

We will see more of that. We will see ways to bind variables, parameters to functions and so on. So, these are heavily used in C++ 11. So, I though at a basic level the support is also available for C++ 03, I chose not to discuss them here because it would be better to discuss it once when we discuss C++ 11 standard library as a whole.

(Refer Slide Time: 34:16)



So, that brings us to the closure of the discussion on the C++ standard library. I have obviously, while I have tried to summarize the containers extensively, because I feel the containers, iterators and associated algorithms are the most useful. But we have been able to glimpse through only some of the algorithms and functional, particularly, partly.

And there are number of other, even besides IO related ones, are a number of other components as well, which are less frequently used. Once you learn the style of STL based standard library use, I think any other component you will be able to learn by yourself very easily. Thank you very much for your attention. This brings us more or less to the closure of C++ 03 discussions. The remaining three weeks, we will really spend on the modern part of C++ which will be C++ 11 and at times C++ 14, C++ 17 and so on.