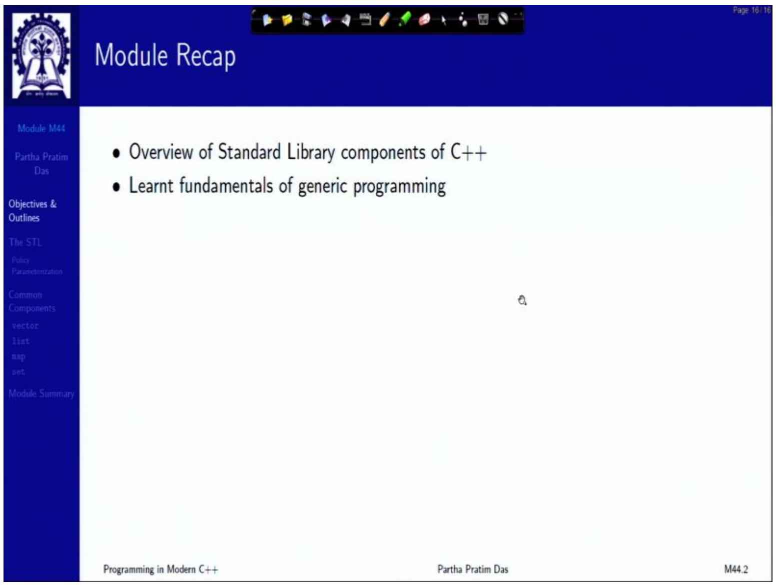
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 44
C++ Standard Library: Part 2 (STL)

Welcome to Programming in Modern C++. We are in week 9 and we are going to discuss Module 44.

(Refer Slide Time: 00:34)

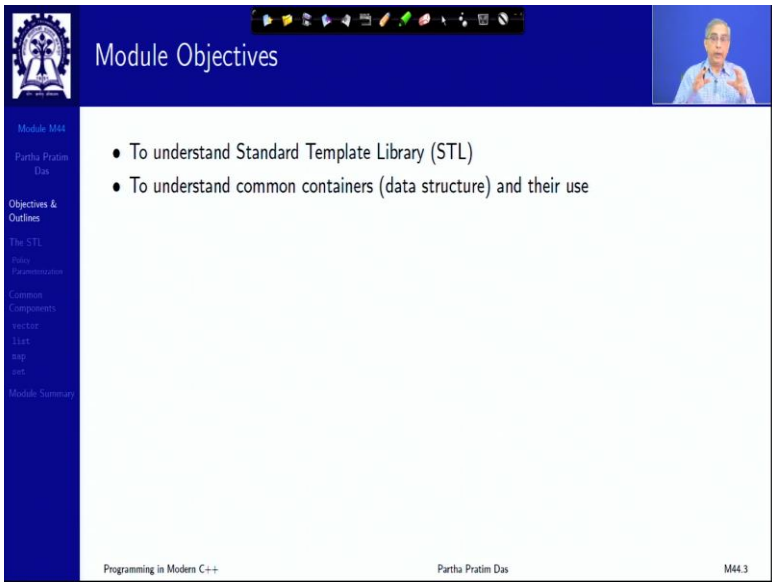


Module Recap

- Overview of Standard Library components of C++
- Learnt fundamentals of generic programming

In the last module, we introduced the standard library overview of C++. And most importantly, we have learnt the fundamentals of generic programming, how to do Meta programming in C++ which will be the backbone for the standard template library.

(Refer Slide Time: 00:54)


Module Objectives

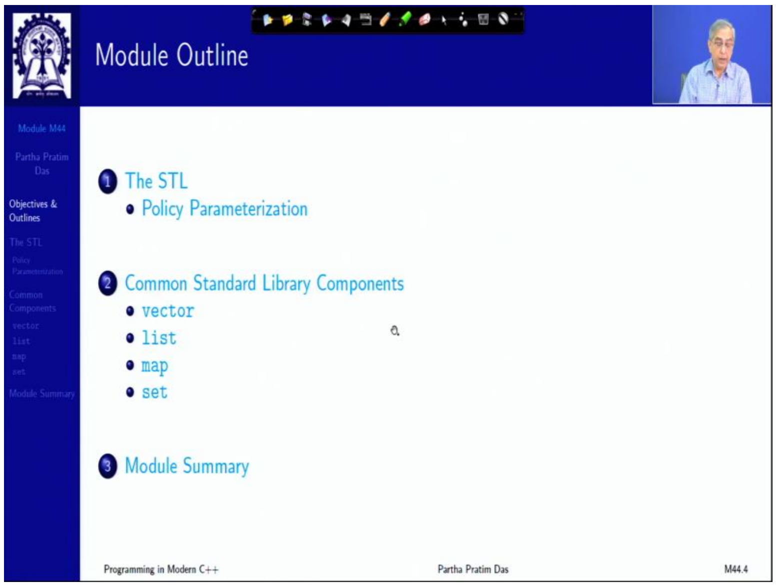
- To understand Standard Template Library (STL)
- To understand common containers (data structure) and their use

Module M44
Partha Pratim Das
Objectives & Outlines
The STL
Policy Parameterization
Common Components
vector
list
map
set
Module Summary

Programming in Modern C++ Partha Pratim Das M44.3

So, in this module, we will understand the standard template library better, and specifically, take a look at some common container or data structures and their use.

(Refer Slide Time: 01:06)

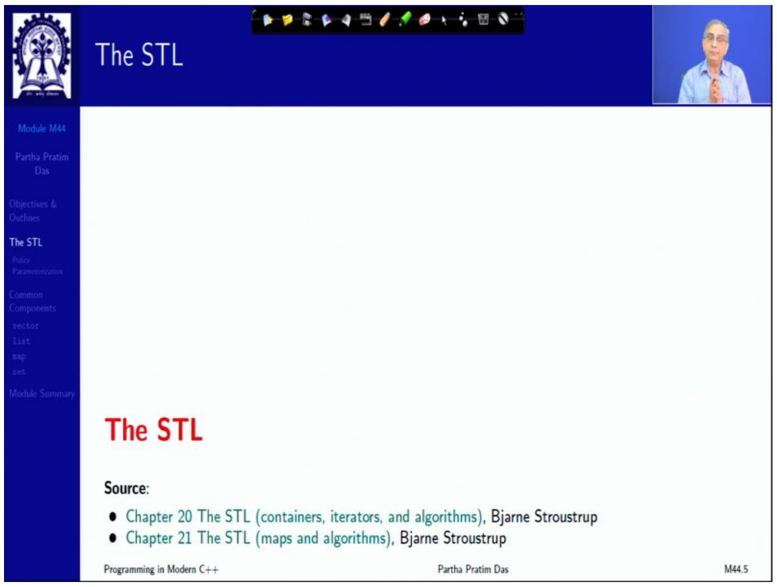

Module Outline

1. The STL
   - Policy Parameterization
2. Common Standard Library Components
   - `vector`
   - `list`
   - `map`
   - `set`
3. Module Summary

Programming in Modern C++ Partha Pratim Das M44.4

So, this will be the outline and will be available on the left panel.

(Refer Slide Time: 01:12)



The STL

Module M44

Partha Pratim Das

Objectives & Outlines

The STL

Policy Parameterization

Common Components

vector

list

map

set

Module Summary

The STL

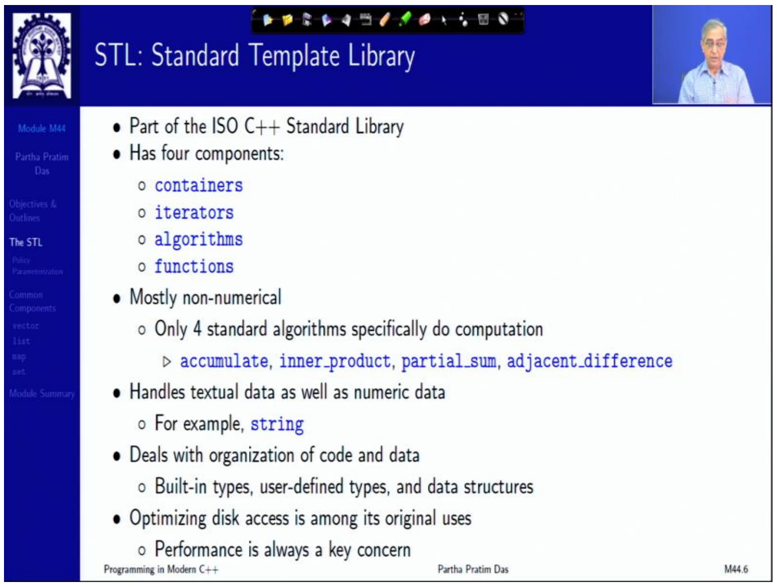
Source:

- Chapter 20 The STL (containers, iterators, and algorithms), Bjarne Stroustrup
- Chapter 21 The STL (maps and algorithms), Bjarne Stroustrup

Programming in Modern C++ Partha Pratim Das M44.5

So, first let me formally introduce STL, Standard Template Library and it is customary to call it the STL. You will soon realize why.

(Refer Slide Time: 01:23)



STL: Standard Template Library

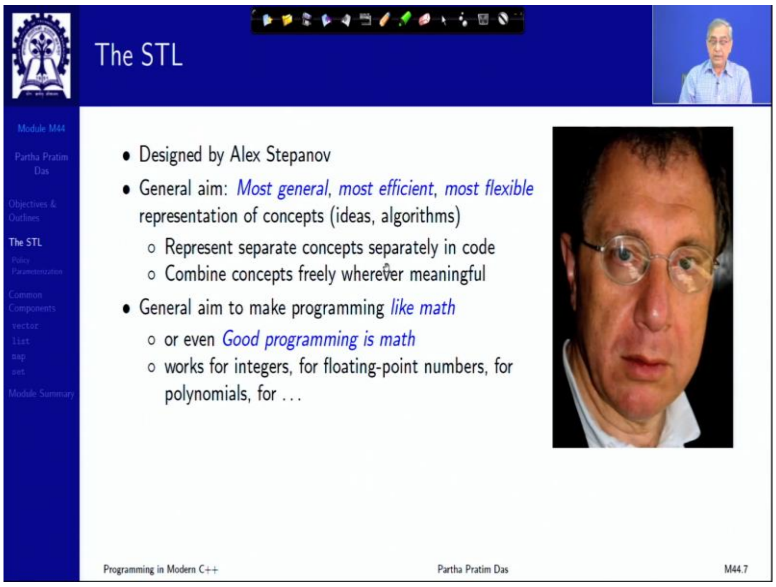
- Part of the ISO C++ Standard Library
- Has four components:
  - containers
  - iterators
  - algorithms
  - functions
- Mostly non-numerical
  - Only 4 standard algorithms specifically do computation
    - accumulate, inner_product, partial_sum, adjacent_difference
- Handles textual data as well as numeric data
  - For example, string
- Deals with organization of code and data
  - Built-in types, user-defined types, and data structures
- Optimizing disk access is among its original uses
  - Performance is always a key concern

Programming in Modern C++ Partha Pratim Das M44.6

So, STL is a part of ISO C++ Standard Library. STL as such has four components; containers, iterators, algorithms. We have talked about these three and functions which are basically function objects. So, these are the primary components of STL, but we do not say that STL is such a, you know, well-defined subset, because we will see the use of these components, in turn, in almost every other component of the C++ Standard Library.

Now, the STL is mostly non-numerical. There are only four standard algorithms that compute. These are the four standard algorithms. There is a component called numeric which does that. It can handle textual as well as numeric data. For example, it can most of STL can work with say, things like string or user defined data type. So, it works with built in data type, user defined types, data structures and so on so forth.

(Refer Slide Time: 02:30)



The STL

- Designed by Alex Stepanov
- General aim: *Most general*, *most efficient*, *most flexible* representation of concepts (ideas, algorithms)
  - Represent separate concepts separately in code
  - Combine concepts freely wherever meaningful
- General aim to make programming *like math*
  - or even *Good programming is math*
  - works for integers, for floating-point numbers, for polynomials, for . . .

Module M44

Partha Pratim Das

Objectives & Outlines

The STL

Policy Parameterization

Common Components

vector

list

map

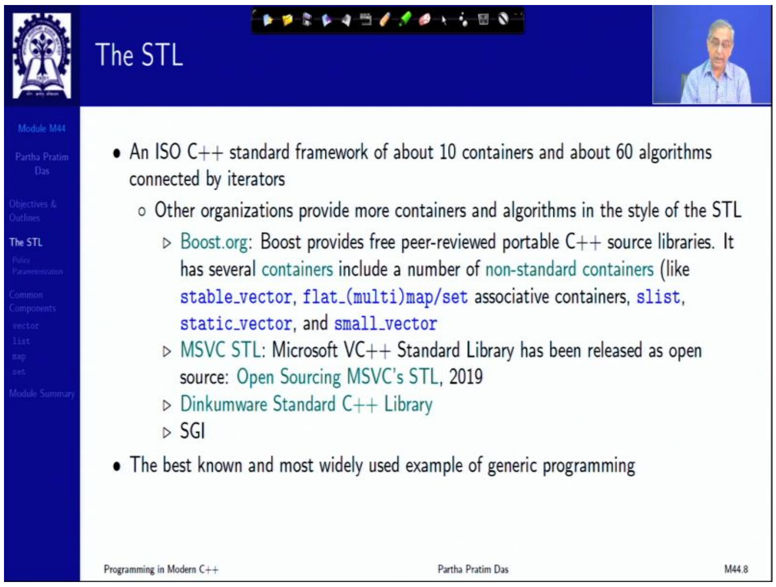
set

Module Summary

Programming in Modern C++ Partha Pratim Das M44.7

This amazing piece of the library is an outcome of about 15 years of research by this great computer scientist Alex Stepanov, who offered it to the standards committee and was subsequently adopted in very well in C++ to make things most generic, most efficient and most flexible.

(Refer Slide Time: 02:57)



The STL

- An ISO C++ standard framework of about 10 containers and about 60 algorithms connected by iterators
  - Other organizations provide more containers and algorithms in the style of the STL
    - Boost.org: Boost provides free peer-reviewed portable C++ source libraries. It has several containers include a number of non-standard containers (like `stable_vector`, `flat_(multi)map/set` associative containers, `slist`, `static_vector`, and `small_vector`
    - MSVC STL: Microsoft VC++ Standard Library has been released as open source: Open Sourcing MSVC's STL, 2019
    - Dinkumware Standard C++ Library
    - SGI
- The best known and most widely used example of generic programming

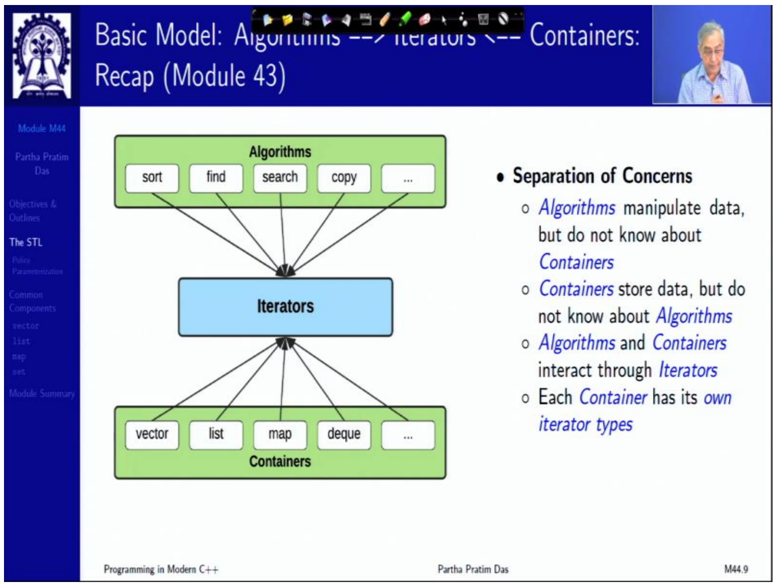
Programming in Modern C++ Partha Pratim Das M44.8

Now your question would be that, is the ISO standardized STL or C++ standard library? Is it the only one that exists in the world? The answer is, of course, no. There are several other implementations of the similar ideas of standard template library. The most common being Boost, which is an open source, free peer reviewed, portable C++ source library.

The big advantage of Boost is, Boost does a lot of experiments for the future generations of C++. Many of the things that we currently have in C++ standard library has evolved or even the language features have evolved from experiments in Boost. So, you can consider contributing to Boost as well.

There are several others which are commercial like Microsoft Visual C++ STL. Then others and so on. But most widely, the best-known and most widely used example of generic programming happens in the ISO standardized standard template library only. That is the reason we call it the STL. It is not a STL; it is the STL that it uses.

(Refer Slide Time: 04:08)



Basic Model: Algorithms --> Iterators <-- Containers: Recap (Module 43)

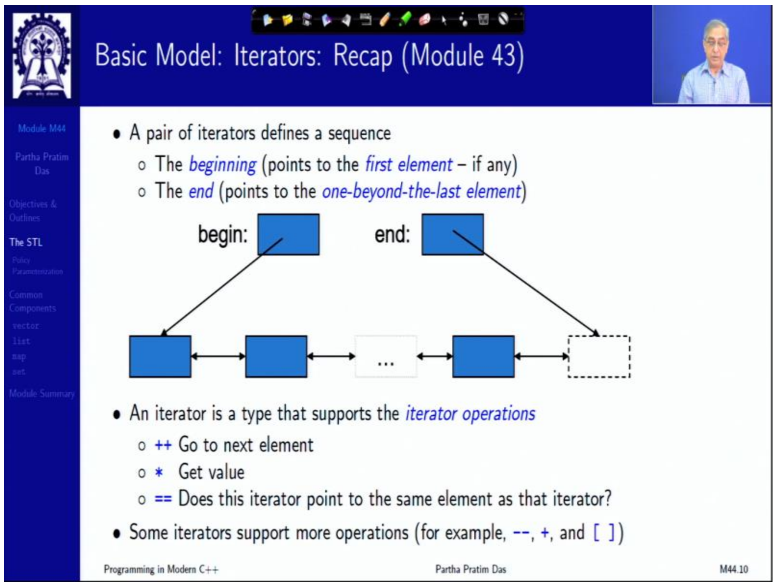
- Separation of Concerns
  - *Algorithms* manipulate data, but do not know about *Containers*
  - *Containers* store data, but do not know about *Algorithms*
  - *Algorithms* and *Containers* interact through *Iterators*
  - Each *Container* has its *own iterator types*

Module M44 | Partha Pratim Das | Objectives & Outlines | The STL | Policy Parameterization | Common Components | vector | list | map | set | Module Summary

Programming in Modern C++ Partha Pratim Das M44.9

We have talked about the basic model, that the basic model is to, you know, separate the concerns of an algorithm and the container, separately. The container and algorithm interact through the concept of iterators which go over the data structure in a physically or virtually linearized form, to get give the algorithm the next data element as and when it needs. And starts from the beginning, goes up to the end, or wherever you ask it to start and wherever you ask it to end.

(Refer Slide Time: 04:47)



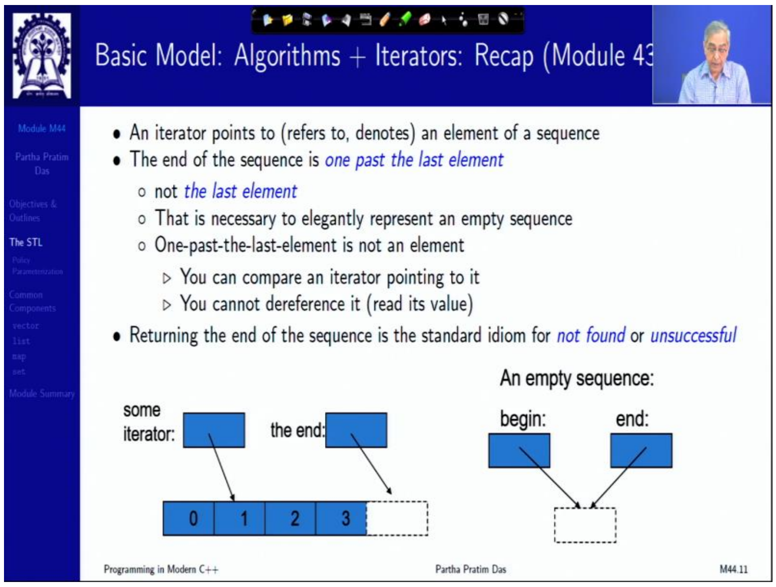
Basic Model: Iterators: Recap (Module 43)

- A pair of iterators defines a sequence
  - The *beginning* (points to the *first element* – if any)
  - The *end* (points to the *one-beyond-the-last element*)

- An iterator is a type that supports the *iterator operations*
  - ++ Go to next element
  - * Get value
  - == Does this iterator point to the same element as that iterator?
- Some iterators support more operations (for example, --, +, and [ ])

Module M44 | Partha Pratim Das | Objectives & Outlines | The STL | Policy Parameterization | Common Components | vector | list | map | set | Module Summary

Programming in Modern C++ Partha Pratim Das M44.10



Basic Model: Algorithms + Iterators: Recap (Module 43

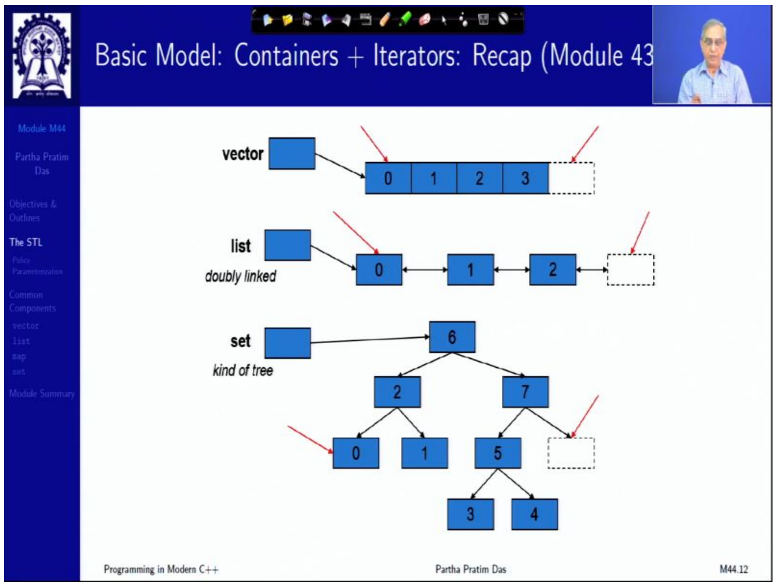
- An iterator points to (refers to, denotes) an element of a sequence
- The end of the sequence is *one past the last element*
  - not *the last element*
  - That is necessary to elegantly represent an empty sequence
  - One-past-the-last-element is not an element
    - You can compare an iterator pointing to it
    - You cannot dereference it (read its value)
- Returning the end of the sequence is the standard idiom for *not found* or *unsuccessful*

Module M44
Partha Pratim Das
Objectives & Outlines
The STL
Policy Parameterization
Common Components
vector
list
map
set
Module Summary

Programming in Modern C++ Partha Pratim Das M44.11

So, with that the basic advantage we noted is that the iterators which are marked by the beginning of a data structure to one beyond the last element of the data structure can be clubbed with the algorithm. So, that the algorithms do not need to know about the containers when those algorithms are coded. They can just rely on the three operators of the iterators, which is ++ to go to the next element, * to get the value of the element and == or != operator to check, if two iterator values are same or not.

(Refer Slide Time: 5:25)



Basic Model: Containers + Iterators: Recap (Module 43

Module M44
Partha Pratim Das
Objectives & Outlines
The STL
Policy Parameterization
Common Components
vector
list
map
set
Module Summary

Programming in Modern C++ Partha Pratim Das M44.12

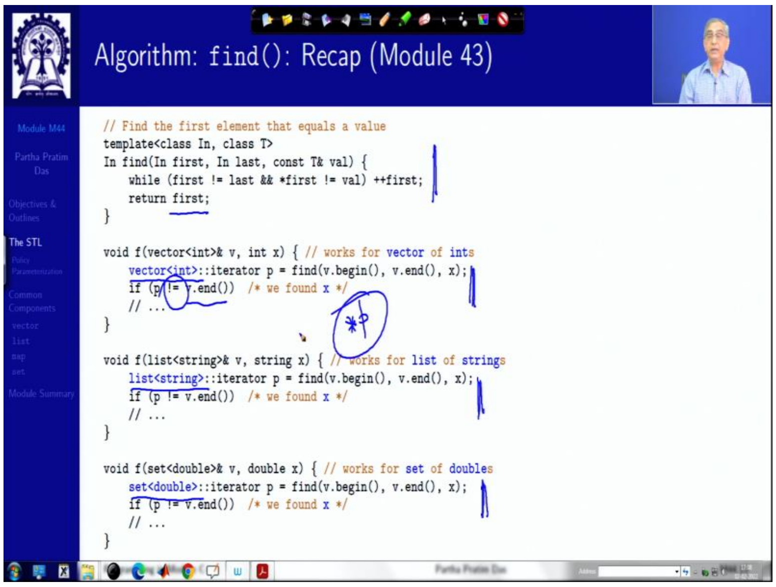
In the same way, the containers also, in turn, do not need to look at, no, the algorithms that are applying on them. Their whole purpose is to support; every container must support the corresponding iterators, which are those three functions. So, how do you conceptualize this on the particular container, if it is a vector, it is an array, it is a sequence container, things happen in a linear manner. So, the zeroth location is the beginning. And one beyond the last location is the end of the iterator.

It is not necessary that it will have to be that. It could be somewhere in the middle also. You could say I will start doing things from 2. I have done something else before that and so on. For a list, this will again be the same. So, only thing that will get different is, the way these three overloaded operators of ++, * and == comparison are implemented by the respective data structure, which is the responsibility of the data structure, responsibility of the container to implement.

So that algorithms can always assume that whatever data container they have, for that the corresponding conceptually same operations are available through syntactically identical code. And this applies; this concept of iteration applies also to non-linear data structure like a set, which is in a binary search tree form.

You can do an in order traversal of the tree to get the order in which the data elements will be given for the iteration or your implementation could choose some other order. But all that you need is there has to be a unique starting point, there has to be a unique end point beyond the last element and there has to be a unique order created out of this non-linear structure.

(Refer Slide Time: 07:22)



Algorithm: find(): Recap (Module 43)

```
// Find the first element that equals a value
template<class In, class T>
In find(In first, In last, const T& val) {
    while (first != last && *first != val) ++first;
    return first;
}

void f(vector<int>& v, int x) { // works for vector of ints
    vector<int>::iterator p = find(v.begin(), v.end(), x);
    if (p != v.end())  /* we found x */
    // ...
}

void f(list<string>& v, string x) { // works for list of strings
    list<string>::iterator p = find(v.begin(), v.end(), x);
    if (p != v.end())  /* we found x */
    // ...
}

void f(set<double>& v, double x) { // works for set of doubles
    set<double>::iterator p = find(v.begin(), v.end(), x);
    if (p != v.end())  /* we found x */
    // ...
}
```

Now with that we saw a find() algorithm, which is very interesting, in that it takes an iterator class In and the element type class in the template. So, then, the find() function takes two iterators, one were to begin, other were to end, defining the range and the value as a constant reference. Because you are finding, so you do not want to change that value.

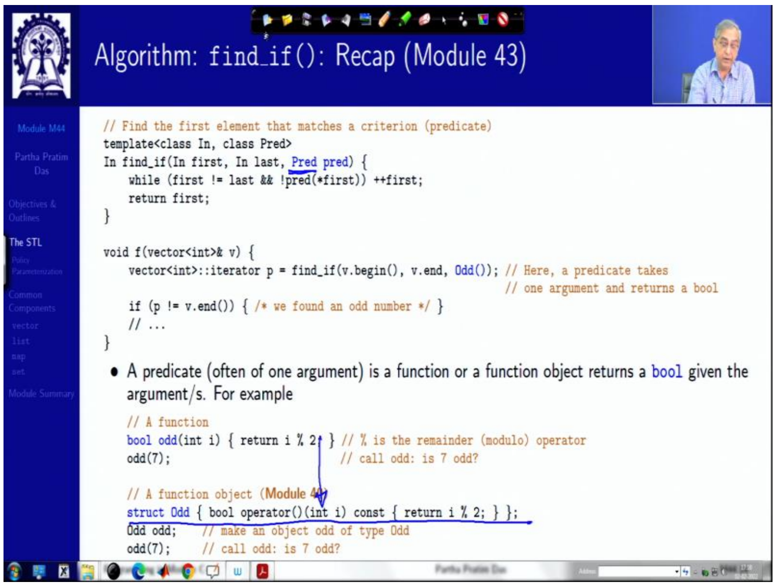
Then the code is extremely simple, which you write in terms of first part checks that there are more elements to check, because first is not equal to last, and the second part checks if the current value *first is the value that you are looking for. If it is not then you go to the next element, as simple as just an iterative find() algorithm. No smart things like binary search or anything being attempted right now.

Now with this, if you have different types of containers with different underlying data types but all of them support the iterator in the same way, the entire code that you write to do the find() is exactly identical. Because the algorithm find() knew that it knew the iterator. So, algorithm find() does not assume the data structure.

The containers whether it is a vector or a list or a double implies has implemented the iterator, did not need to know whether you will find or sort or sum or do whatever. So, it fits in, you can see that except for the blue part everything else in the code is actually identical and this finally returns the iterator value. So, you have to check whether the iterator value is same at the end.

If it is same as the end that means you have crossed the entire data structure and you did not find the value. So, it is a failure. If it is not same as the end, again check the operator that the iterator has to support. If it is not at the end then you must have got the value at the iteration point which is basically the value *p. Because p is your iterator here.

(Refer Slide Time: 09:40)



Algorithm: find_if(): Recap (Module 43)

```
// Find the first element that matches a criterion (predicate)
template<class In, class Pred>
In find_if(In first, In last, Pred pred) {
    while (first != last && !pred(*first)) ++first;
    return first;
}

void f(vector<int>& v) {
    vector<int>::iterator p = find_if(v.begin(), v.end, Odd()); // Here, a predicate takes
                                                                // one argument and returns a bool
    if (p != v.end()) { /* we found an odd number */ }
    // ...
}
```

- A predicate (often of one argument) is a function or a function object returns a bool given the argument/s. For example

```
    // A function
    bool odd(int i) { return i % 2; } // % is the remainder (modulo) operator
    odd(7);                           // call odd: is 7 odd?

    // A function object (Module 44)
    struct Odd { bool operator()(int i) const { return i % 2; } };
    Odd odd;    // make an object odd of type Odd
    odd(7);     // call odd: is 7 odd?
```

So, this is how the generic algorithms are written you will find this find() algorithm in the algorithm component of the STL. You can even generalize this further saying that instead of giving a value I will give a functor here, I will give a predicate here, I will give a, you know, a function pointer or a function object which will check certain property. So, I am saying that you go over this vector and find the first odd element.

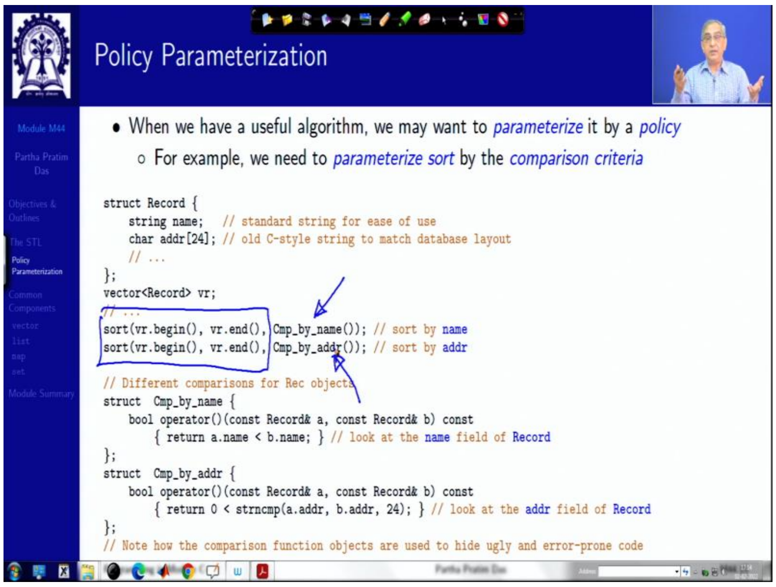
Now this oddness is not a value, oddness is a property. So, I can write a function as a function pointer it to check if a given value is odd or I can write a function object to do that. I can pass either of them. So, what I am passing is I am passing an instance of the Pred class which is basically a function object. So, I pass that.

So, what happens is when I am trying to do the iteration over this data structure, I am just evaluating this function object Pred with the given element. So, which is basically either a call to this function operator, if I pass the functor as we have done here or it is a call to this function if I passed Odd, if I just passed Odd as a function pointer. So, either of that can be used.

And so this will make things even more, more generic things can be done. So, you can write any logic as a part of this function object. And you can find elements which can satisfy variety of different conditions. One catch point to note here is in the earlier form of find, you needed to pass set, the type of the element; here you are not doing that.

Because you have set the type for the functor object. You have set the type for the functor object. So, this functor object here takes the element type. So, since you are calling Odd for checking the, I mean checking as a predicate, it has to take an int. So, it will operate on the int. So, that is the kind of deduction that the compiler would be able to do.

(Refer Slide Time: 12:08)



Policy Parameterization

- When we have a useful algorithm, we may want to *parameterize* it by a *policy*
  - For example, we need to *parameterize sort* by the *comparison criteria*

```
struct Record {
    string name;   // standard string for ease of use
    char addr[24]; // old C-style string to match database layout
    // ...
};
vector<Record> vr;
// ...
sort(vr.begin(), vr.end(), Cmp_by_name()); // sort by name
sort(vr.begin(), vr.end(), Cmp_by_addr()); // sort by addr

// Different comparisons for Rec objects
struct  Cmp_by_name {
    bool operator()(const Record& a, const Record& b) const
        { return a.name < b.name; } // look at the name field of Record
};
struct  Cmp_by_addr {
    bool operator()(const Record& a, const Record& b) const
        { return 0 < strncmp(a.addr, b.addr, 24); } // look at the addr field of Record
};
// Note how the comparison function objects are used to hide ugly and error-prone code
```

Now if you look into these two forms of find, you will find that the basic algorithm is the same. Start, at the beginning keep on checking on the value, keep on going till the end. Now, the way you check that has been changed. In the case of find, the check was by equality. In the case of find if, it is by some other predicate. It could have been equality also. So, such elements in the design of STL is called a policy.

That is generically speaking, I am trying to do find and I have a policy for deciding how I find or what I find. Here is another example on this, which is from your more common domain of sorting. So, I have a record which has say two fields. One is of type string; another is of type character array, which is basically C-style string. One keeps the name, another keeps the address of a person.

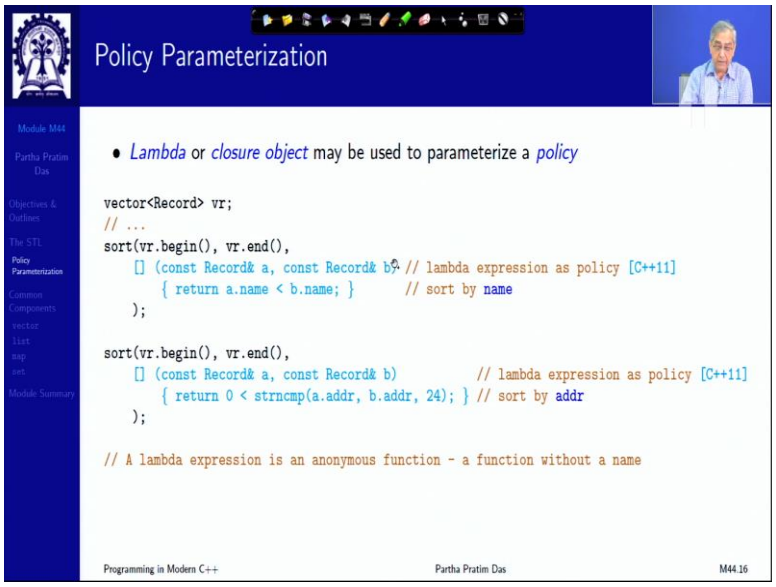
Now, a vector for this record. So, I have of that. So, how do I sort? To sort, I need to, we have seen sort earlier also, but now look at it from the generic programming point of view. To sort, what I need to know? I need to know the range, where to start where to end. So, where to start is the begin iterator, where to end is the end iterator.

And what else you need to know? You need to know how to compare elements. So, that you pass as a function object as a functor. So, I have a functor implemented here as compare by name which overloads the function call operator for a pair of record references and does a comparison by using the compare function of the string type, because these are strings.

Whereas, I also have another functor for comparing addresses which overloads the function call operator again for a pair of records, because to compare I need two records. But it does the comparison using strncmp function of the string.h header of C. Because address is a C-style string. Null terminated array of characters.

So, you can see that, basically, if you look at sort, there is no difference in the sort, only thing that changes is the comparison policy. And based on that the same sort code can be made to use not only on arbitrary data structures but also on arbitrary policies. So, that makes it really really powerful and this is what is known as policy parameterization.

(Refer Slide Time: 15:19)



Policy Parameterization

Module M44

Partha Pratim Das

Objectives & Outlines

The STL

Policy Parameterization

Common Components

vector

list

map

set

Module Summary

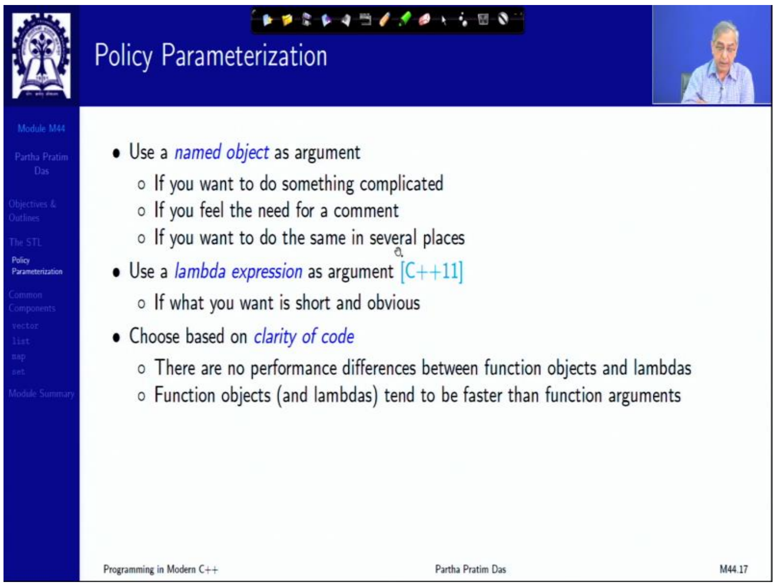
- *Lambda* or *closure object* may be used to parameterize a *policy*

```
vector<Record> vr;
// ...
sort(vr.begin(), vr.end(),
    [] (const Record& a, const Record& b) // lambda expression as policy [C++11]
        { return a.name < b.name; }       // sort by name
    );

sort(vr.begin(), vr.end(),
    [] (const Record& a, const Record& b)           // lambda expression as policy [C++11]
        { return 0 < strncmp(a.addr, b.addr, 24); } // sort by addr
    );

// A lambda expression is an anonymous function - a function without a name
```

Programming in Modern C++ Partha Pratim Das M44.16

So, many of the algorithms available in STL has the policy parameterization, often they have a default policy. But there is a provision to parameterize the policy, as you want. There are some compact, more compact ways of doing that as well.

(Refer Slide Time: 15:37)



Policy Parameterization

Module M44

Partha Pratim Das

Objectives & Outlines

The STL

Policy Parameterization

Common Components

vector

list

map

set

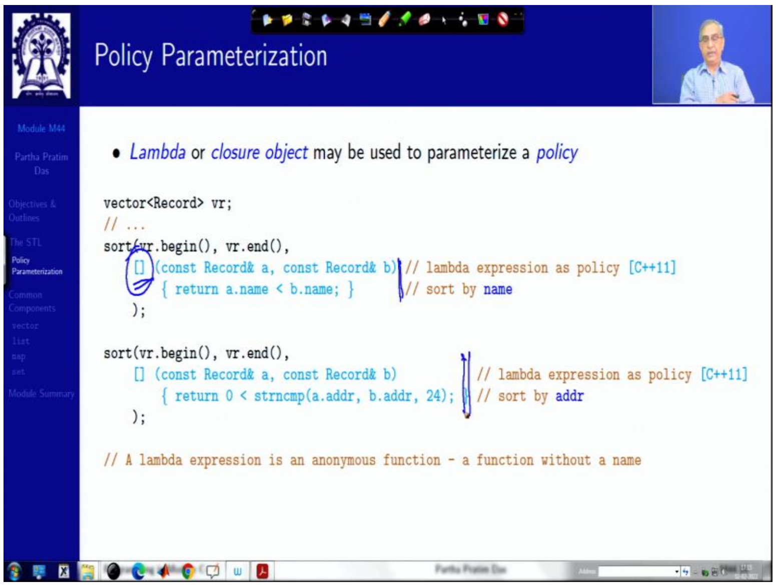
Module Summary

- Use a *named object* as argument
  - If you want to do something complicated
  - If you feel the need for a comment
  - If you want to do the same in several places
- Use a *lambda expression* as argument [C++11]
  - If what you want is short and obvious
- Choose based on *clarity of code*
  - There are no performance differences between function objects and lambdas
  - Function objects (and lambdas) tend to be faster than function arguments

Programming in Modern C++ Partha Pratim Das M44.17

I have just shown as example here but we will come back to this heavily when we do C++11.

(Refer Slide Time: 15:40)



Policy Parameterization

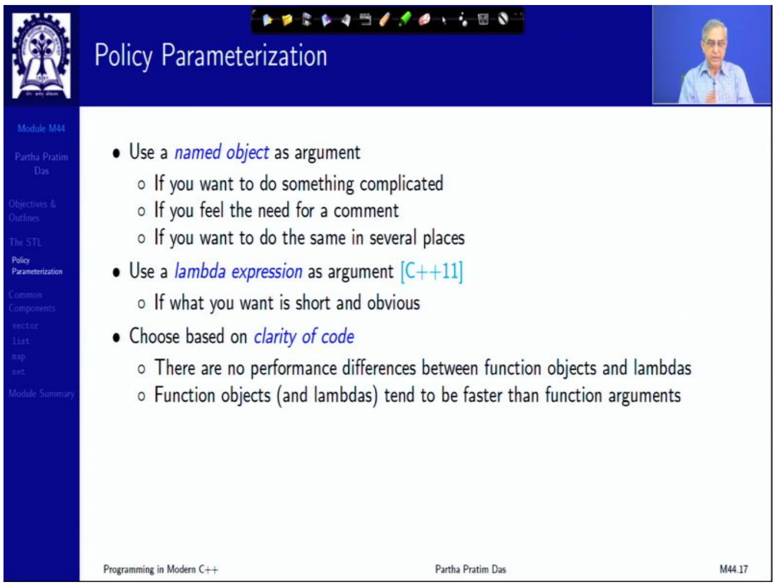
- *Lambda* or *closure object* may be used to parameterize a *policy*

```
vector<Record> vr;
// ...
sort(vr.begin(), vr.end(),
    [] (const Record& a, const Record& b) // lambda expression as policy [C++11]
        { return a.name < b.name; }       // sort by name
    );

sort(vr.begin(), vr.end(),
    [] (const Record& a, const Record& b)         // lambda expression as policy [C++11]
        { return 0 < strncmp(a.addr, b.addr, 24); } // sort by addr
    );

// A lambda expression is an anonymous function - a function without a name
```

Here, what I am using is I am using the concept of an anonymous function, a function object which does not have a name. So, the entire code is written here and the all of that struct, bool operator, parenthesis, all that are shorted in the form of a pair of square brackets which is called a lambda or a closure object in C++11, you will see more of that. So, I can also write the policies in this form.

Certainly, the advantage is that if the policy is simple as it is here then writing it inside the sort call makes it easier to understand, makes it easier to follow as to what exactly are you doing. Of course, if it is a big code, you will not be doing this, you will not be using lambda. You will be using a named object, as we did earlier. But lambdas are available for doing this. We will see more of that. Do not worry about not understanding lambda. You will you will have ample time to understand lambda in the subsequent module.

(Refer Slide Time: 16:53)



Policy Parameterization

Module M44

Partha Pratim Das

Objectives & Outlines

The STL

Policy Parameterization

Common Components

vector

list

map

set

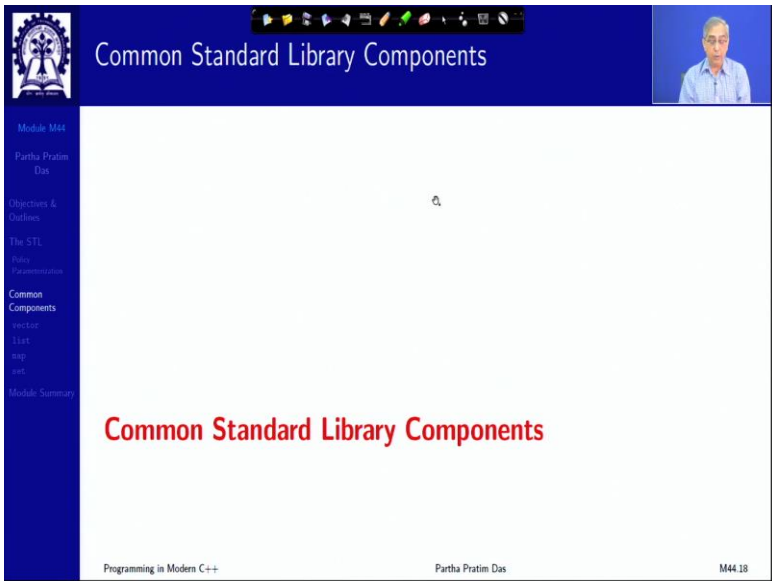
Module Summary

- Use a *named object* as argument
  - If you want to do something complicated
  - If you feel the need for a comment
  - If you want to do the same in several places
- Use a *lambda expression* as argument [C++11]
  - If what you want is short and obvious
- Choose based on *clarity of code*
  - There are no performance differences between function objects and lambdas
  - Function objects (and lambdas) tend to be faster than function arguments

Programming in Modern C++ Partha Pratim Das M44.17

So, policy parameterization, in general, can be done through named objects as we have done or through lambda expressions, depending on whether you want to reuse the same policy in multiple places you will use a named object or if it is if you need to have a lot of comments there in or you know it is quite a complicated logic and so on otherwise you just use a lambda.

(Refer Slide Time: 17:18)



Common Standard Library Components

Module M44

Partha Pratim Das

Objectives & Outlines

The STL

Policy Parameterization

Common Components

vector

list

map

set

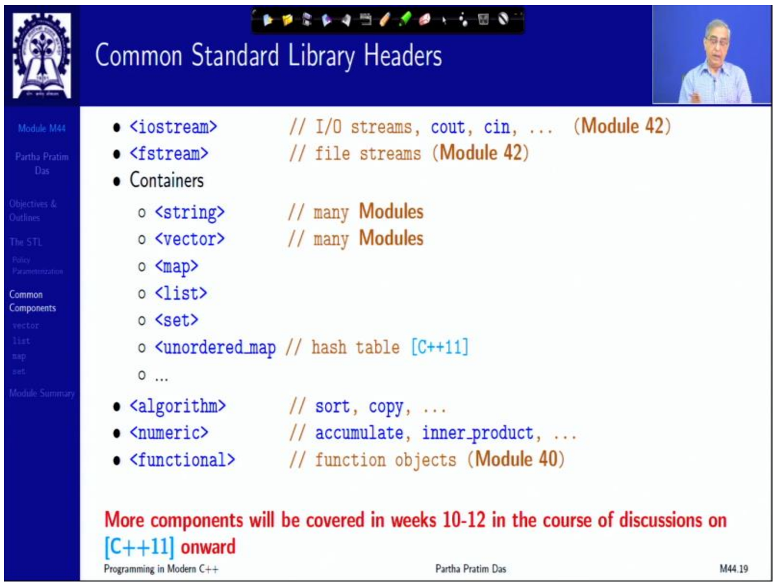
Module Summary

Common Standard Library Components

Programming in Modern C++ Partha Pratim Das M44.18

So, with that let us take a look at the common standard library components.

(Refer Slide Time: 17:24)



Common Standard Library Headers

```
• <iostream>        // I/O streams, cout, cin, ...  (Module 42)
• <fstream>         // file streams (Module 42)
• Containers
   ○ <string>       // many Modules
   ○ <vector>       // many Modules
   ○ <map>
   ○ <list>
   ○ <set>
   ○ <unordered_map // hash table [C++11]
   ○ ...
• <algorithm>       // sort, copy, ...
• <numeric>         // accumulate, inner_product, ...
• <functional>      // function objects (Module 40)
```

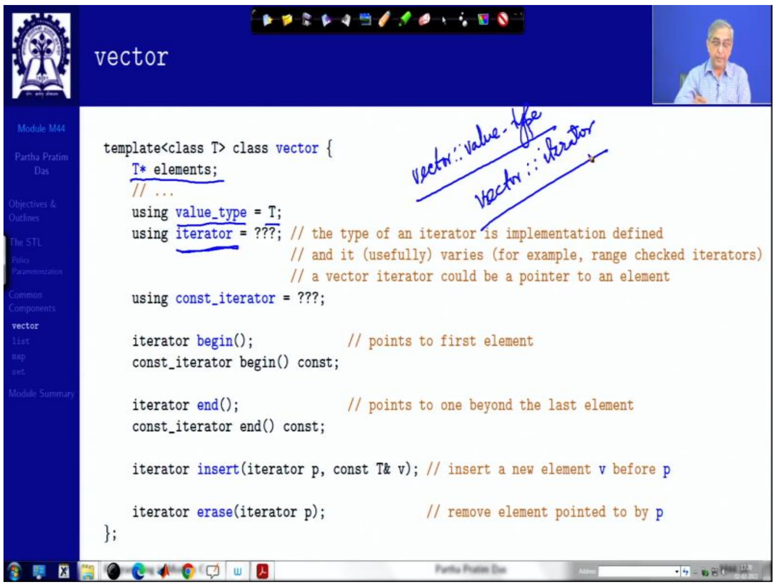
More components will be covered in weeks 10-12 in the course of discussions on [C++11] onward

Programming in Modern C++ Partha Pratim Das M44.19

So, this is just a very very small part of the standard library C++ standard library but also it is a fact that possibly these components will cover 80%, 90% of your common usage. iostream, fstream, we have already done. Then there are a number of containers of which also string we have seen in multiple places, we more or less understand.

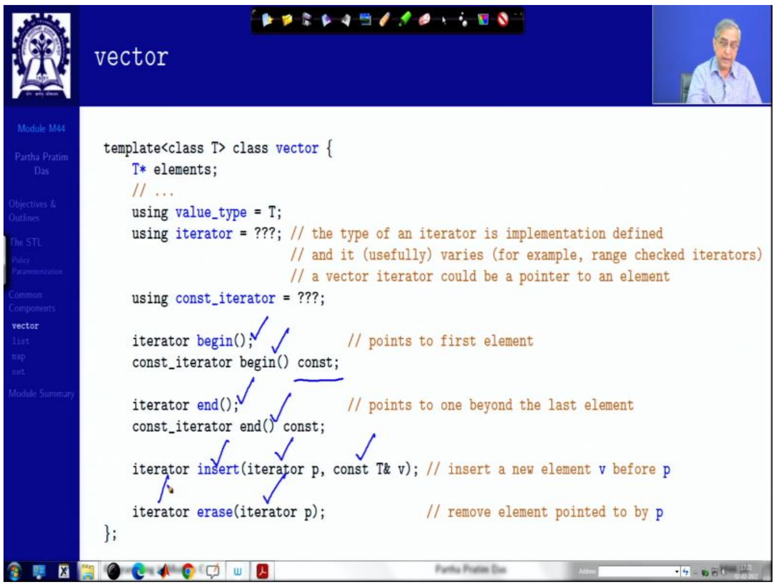
Vector we have seen in multiple places but vector, map, list, these we will briefly discuss in this module, to give you a better idea of the uniformity that exist between the containers of C++ standard library. Then there are other components which are very very useful. The algorithms which give you all sorts of common algorithms, the numeric and the functional. And we will have more components coming when you do C++11 from the next week onwards.

(Refer Slide Time: 18:17)



vector

```
template<class T> class vector {
    T* elements;
    // ...
    using value_type = T;
    using iterator = ???; // the type of an iterator is implementation defined
                          // and it (usefully) varies (for example, range checked iterators)
                          // a vector iterator could be a pointer to an element
    using const_iterator = ???;

    iterator begin();           // points to first element
    const_iterator begin() const;

    iterator end();             // points to one beyond the last element
    const_iterator end() const;

    iterator insert(iterator p, const T& v); // insert a new element v before p

    iterator erase(iterator p);            // remove element pointed to by p
};
```



vector

```
template<class T> class vector {
    T* elements;
    // ...
    using value_type = T;
    using iterator = ???; // the type of an iterator is implementation defined
                          // and it (usefully) varies (for example, range checked iterators)
                          // a vector iterator could be a pointer to an element
    using const_iterator = ???;

    iterator begin();           // points to first element
    const_iterator begin() const;

    iterator end();             // points to one beyond the last element
    const_iterator end() const;

    iterator insert(iterator p, const T& v); // insert a new element v before p

    iterator erase(iterator p);            // remove element pointed to by p
};
```

So, let us look at what is the structure of a vector declaration. This is not the complete vector class in the standard template library but this is the representative part. So, the philosophy is, since there are multiple containers, the philosophy is to have as much of uniformity as possible.

So, a number of containers have a number of functionality which is common, particularly, you will need, you will have element type. You cannot have a container where you do not have element type. This is not possible. You will have, you will need iterators to write code in the model that we have shown.

So, there has to be iterators. If we have iterators, there has to be begin and end of the iterator. Since it is a container, there should be some way to insert, some way to remove element and, so on. So, these are standardized to the extent possible in terms of syntax, semantics, even name, so that you do not really need to memorize a lot of things.

If you understand it for one data structure, you will be able to guess most of the member functions and types of another container. You know almost with 90% certainty, rest of it you look up the manual. So, all that we need is for a, let me just explain vector then will go quickly over the rest.

So, you need the actual underlying container which is kind of T is the element type. So, elements is a pointer to T. So, it is an array, as you can understand. So, vector has a type definition, typedef type alias for value type. So, if I do vector::value_type, I will be able to know what is the element type. So, that is defined as T. So, that is that is a very easy way of doing things. So, at any point you can know this. And this is uniform for most of the containers. You need to use an iterator, so you are saying using iterator.

Now, what is the type of that iterator? I am not, I did not try to write this because this is a highly implementation dependent. This is not standardized because this needs lot of optimizations based on the particular machine on which the compiler will target. So, this is implementation-defined feature.

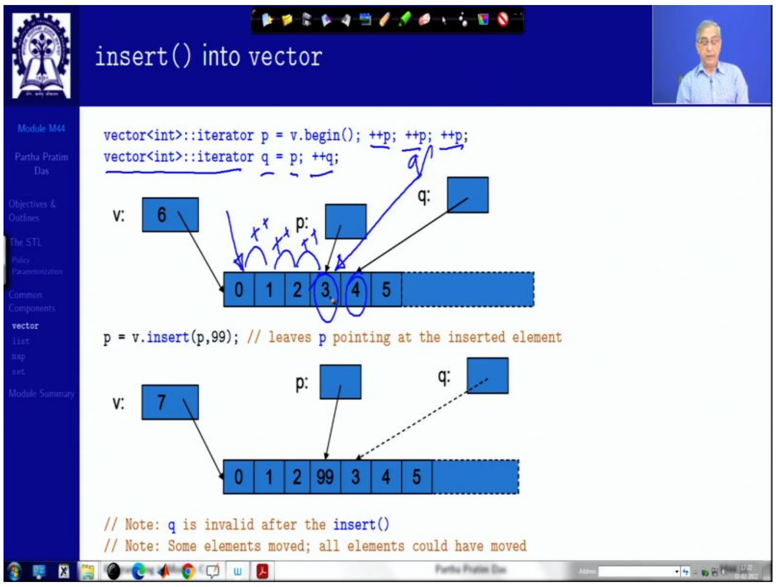
So, you do not need to really bother about what that whole type expression is. All that you need to know is it has a name iterator. So, you are looking, you are working with a vector, your iterator type will be vector::iterator. The nice thing about the uniformity is, if you are working with list, your iterator type is list::iterator.

If you are working with a map, it is map::iterator and, so on. Then your iterator could be a constant iterator. That is it does not allow you to make changes in the code, I am sorry, in the container values cannot be changed. So, you have a const iterator which is also uniform.

And then you have the standard member function. You have a begin for the non-const iterator, you have a begin with for the const iterator. Naturally, with the const iterator it has to be a constant member function. And similarly, you have an insert which takes an iterator and a value and returns you an iterator. That is you are inserting into the vector.

So, you give the iterator to a point where you want to insert. So, it will be inserted before that and that iterator will be returned. Similarly, you do erase by giving an iterator and getting back the iterator after the erase. So, you can see that it is the whole target is to do a very very uniform design and as we will come to, in the next data container type, you will see that how similar the entire design is.

(Refer Slide Time: 22:49)



insert() into vector

```
vector<int>::iterator p = v.begin(); ++p; ++p; ++p;
vector<int>::iterator q = p; ++q;
```

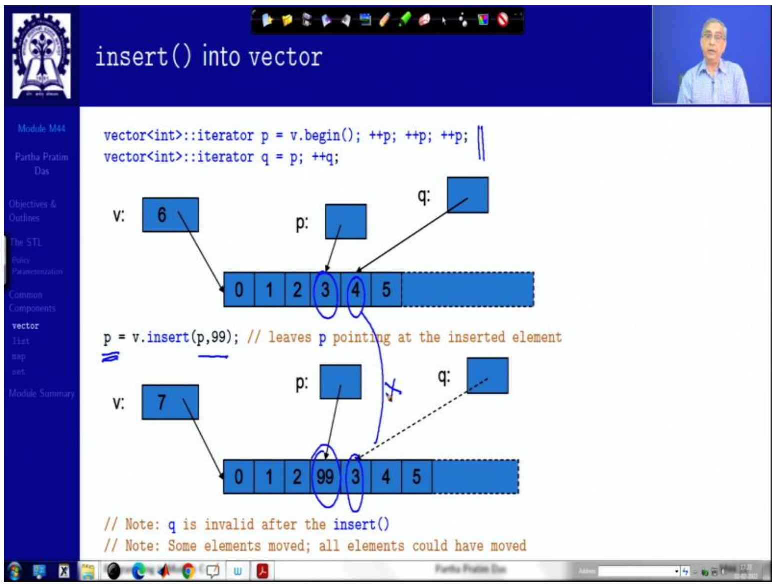
```
p = v.insert(p,99); // leaves p pointing at the inserted element
```

```
// Note: q is invalid after the insert()
// Note: Some elements moved; all elements could have moved
```

But before that let me just quickly take you through some of the differences in semantics that may rise, give rise to this. Syntactically they are very similar. So, let us say, I am talking about insert of vector. So, this is this is a vector that I have and. So, I this obviously is my iterator type p as a vector of integer. So, vector<int>::iterator p. So, begin is here. I am starting at this point.

Now, so p, I am sorry, let us, let me mark it separately. So, p, v.begin will take me here which is p. Now, I do three plus p’s. 1 ++, 1 ++, 1 ++. Three times I increment the iterator which means p will now point to this element 3. I define another iterator q same as p. So, q will point here. But I do ++q. So, q points to the next element. So, this is how given this code I come to this iterator positions.

(Refer Slide Time: 24:12)



insert() into vector

```
vector<int>::iterator p = v.begin(); ++p; ++p; ++p;
vector<int>::iterator q = p; ++q;
```

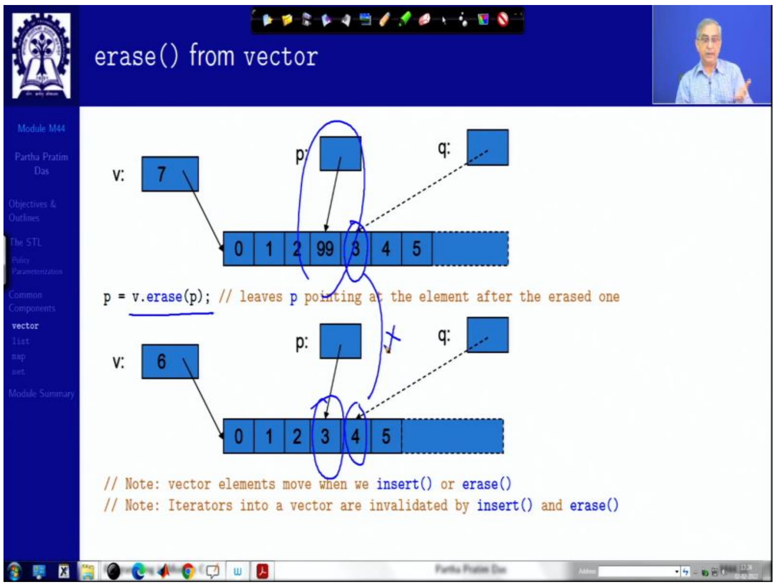
```
p = v.insert(p,99); // leaves p pointing at the inserted element
```

```
// Note: q is invalid after the insert()
// Note: Some elements moved; all elements could have moved
```

Now, I do an insert with p of a value 99. As I do the insert, what will happen? It will be get inserted before the value that p is pointing to that is the semantics. And all values to the will have to be shifted. So, all values will get shifted. So, p will get returned, whatever is returned is assigned to p.

So, that is the position of the new element. You get 99. But as all elements get shifted q now points to 3, it was pointing to 4. So, q actually has become invalid. It is pointing to something which is wrong. So, remember, if you insert in a vector, your pointers, your iterators, other iterators will get invalid.

(Refer Slide Time: 25:04)



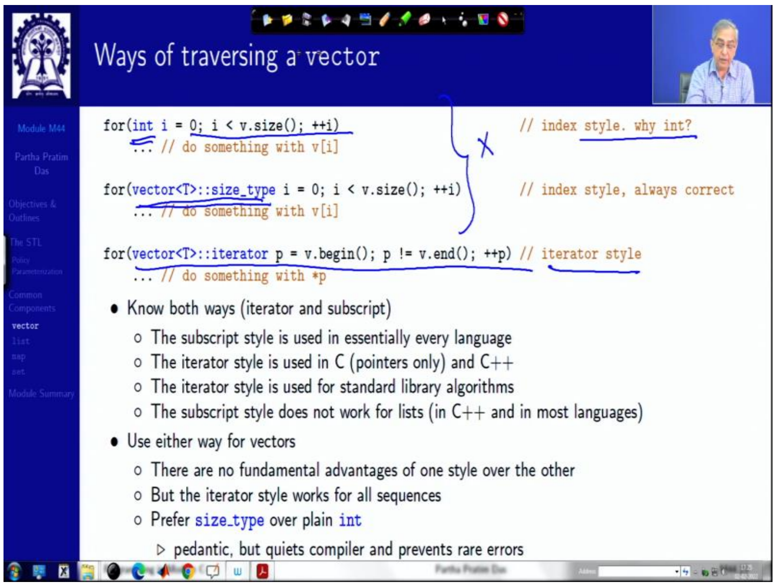
erase() from vector

```
p = v.erase(p); // leaves p pointing at the element after the erased one
```

```
// Note: vector elements move when we insert() or erase()
// Note: Iterators into a vector are invalidated by insert() and erase()
```

Similar, thing you can see in terms of erase. The value immediately that have been inserted here. I am erasing that. So, as I erase, it will get removed values from will be pushed back. It points to 3, but q which was pointing to 3 will now point to 4, again it will get invalid. So, for insert, delete in a vector, the side effect is that other pointers or other iterators will become invalid.

(Refer Slide Time: 25:36)



Ways of traversing a vector

Module M44

Partha Pratim Das

Objectives & Outlines

The STL

Policy Parameterization

Common Components

vector

list

map

set

Module Summary

```
for(int i = 0; i < v.size(); ++i)                               // index style. why int?
    ... // do something with v[i]

for(vector<T>::size_type i = 0; i < v.size(); ++i)              // index style, always correct
    ... // do something with v[i]

for(vector<T>::iterator p = v.begin(); p != v.end(); ++p) // iterator style
    ... // do something with *p
```

X

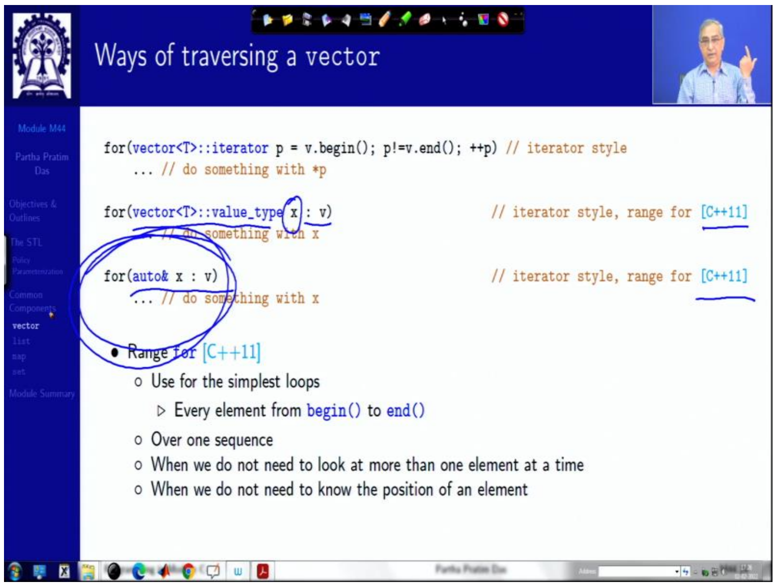
- Know both ways (iterator and subscript)
  - The subscript style is used in essentially every language
  - The iterator style is used in C (pointers only) and C++
  - The iterator style is used for standard library algorithms
  - The subscript style does not work for lists (in C++ and in most languages)
- Use either way for vectors
  - There are no fundamental advantages of one style over the other
  - But the iterator style works for all sequences
  - Prefer size_type over plain int
    - ▷ pedantic, but quiets compiler and prevents rare errors

Partha Pratim Das

Now, see, if I have a vector, what are the different ways I can traverse, it obviously there could be multiple that we have already seen. One is I can just use the index, very simple. I can use the index using a variable which is of type int. It is ok but not very advisable. Because the implementer may have implemented the index using some other type, say, unsigned int. So, what vector does, in fact, almost all containers do that is, they have a type variable called size_type which gives you the type of the size variables in for that container.

So, it is better, if you are using index, it is better to use size_type. Or better still I could have used an iterator, simply. I am doing the same thing just traversing it. So, I could have used any one of this. Of that the iterator form is what we will slowly move more towards, because in both of these, the actual code is closely bound to the fact that it is a vector. I cannot have kind of, I cannot have do ++i and go to the next element in the list. So, I would prefer to go by the iterator's style.

(Refer Slide Time: 27:08)



Ways of traversing a vector

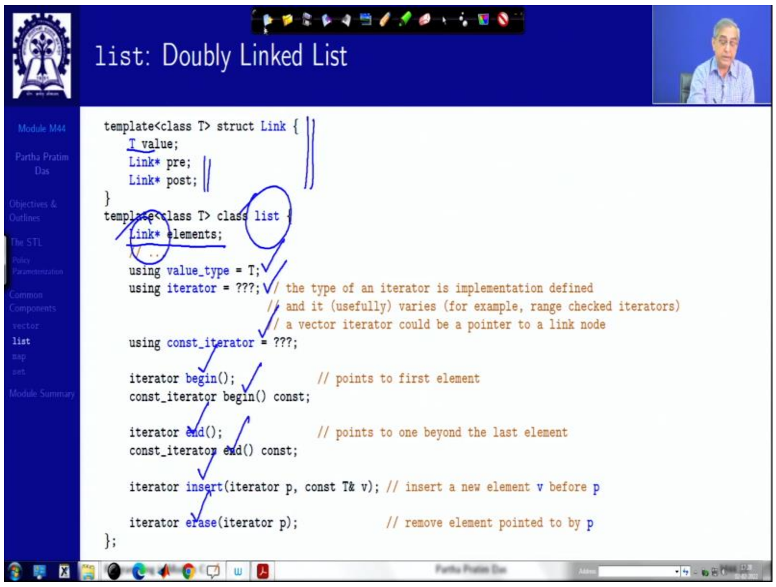
```
for(vector<T>::iterator p = v.begin(); p!=v.end(); ++p) // iterator style
    ... // do something with *p

for(vector<T>::value_type x : v)                // iterator style, range for [C++11]
    ... // do something with x

for(auto& x : v)                                // iterator style, range for [C++11]
    ... // do something with x
```

- Range for [C++11]
  - Use for the simplest loops
    - Every element from begin() to end()
  - Over one sequence
  - When we do not need to look at more than one element at a time
  - When we do not need to know the position of an element

There are other, with this iterator style; there are other styles which will be coming in C++11, particularly, when you are doing an entire data structure traversal. For example, you can say the value_type is x, value type we have already explained. And just colon v. In which case what it will do? It will start from the very beginning of the container and go up to the end. This is called a range kind of support.

You can even simplify it even further by just saying it is auto&. You know what auto& means we will come to C++ eleven, but just to give you a glimpse that iterator style is really really strong and you can you can see in here there is no dead container, no type, nothing is mentioned. It is just the variable given which is a vector of int in our case. But in given any other variable corresponding to any other container of any other underlying type, this will also work.

(Refer Slide Time: 28:11)



list: Doubly Linked List

Module M44

Partha Pratim Das

Objectives & Outlines

The STL

Policy Parameterization

Common Components

vector

list

map

set

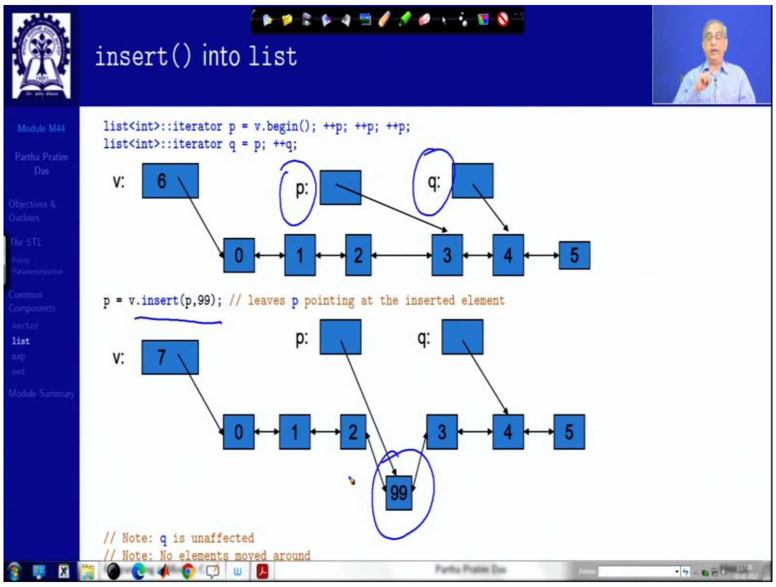
Module Summary

```
template<class T> struct Link {
    T value;
    Link* pre;
    Link* post;
}
template<class T> class list {
    Link* elements;
    // ...
    using value_type = T;
    using iterator = ???; // the type of an iterator is implementation defined
                          // and it (usefully) varies (for example, range checked iterators)
                          // a vector iterator could be a pointer to a link node
    using const_iterator = ???;

    iterator begin();           // points to first element
    const_iterator begin() const;

    iterator end();             // points to one beyond the last element
    const_iterator end() const;

    iterator insert(iterator p, const T& v); // insert a new element v before p

    iterator erase(iterator p);              // remove element pointed to by p
};
```

So, let us consider a doubly linked list. Just to see the parallel. This is the node structure which is trivial. T is the value. There are two links for the double link. Naturally, you have a link pointer as the header. This time it is the header. Now, you have the same value type because that is the underlying type. The contain, the iterator and constant iterator types will be available.

So, will be the begin for non-constant and constant iterator, end for non-constant and constant iterator. Insert in the same way, erase in the same way. We can see exactly, except for this link* and the name list, everything else is same between this and the vector. Because conceptually they are all same. So, the code is generic and can be worked in a generic way.

(Refer Slide Time: 29:13)



insert() into list

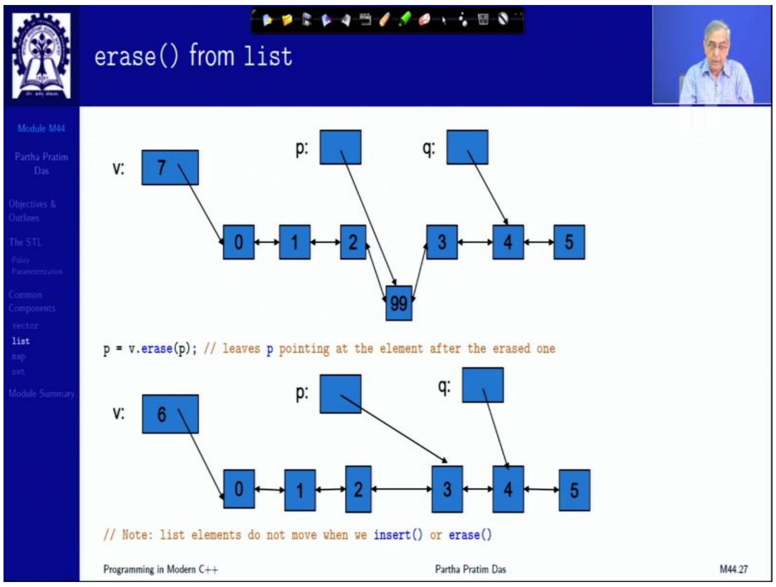
```
list<int>::iterator p = v.begin(); ++p; ++p; ++p;
list<int>::iterator q = p; ++q;
```

```
p = v.insert(p,99); // leaves p pointing at the inserted element
```

```
// Note: q is unaffected
// Note: No elements moved around
```

Of course, depending on the difference of the semantics of the iterator, the side effects will be different. The same situation we are showing here, where we have the same data in the list with p pointing here and q pointing here and I do insert of 99. 99 comes in, but since it is the list, there is no movement of elements. Therefore, q does not become invalid, which was becoming invalid for the case of vector. So, that is for insert, there is a case of erase also worked out.

(Refer Slide Time: 29:48)



erase() from list

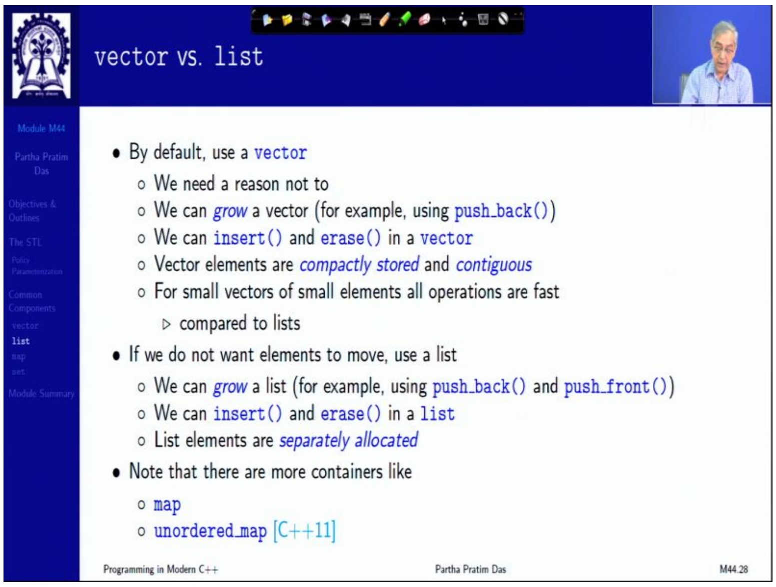
```
p = v.erase(p); // leaves p pointing at the element after the erased one
```

```
// Note: list elements do not move when we insert() or erase()
```

Programming in Modern C++ Partha Pratim Das M44.27

For the case of list, the insert and erase do not invalidate the other iterators but for vector they do. So, there are semantic differences that you will have to understand.

(Refer Slide Time: 30:05)



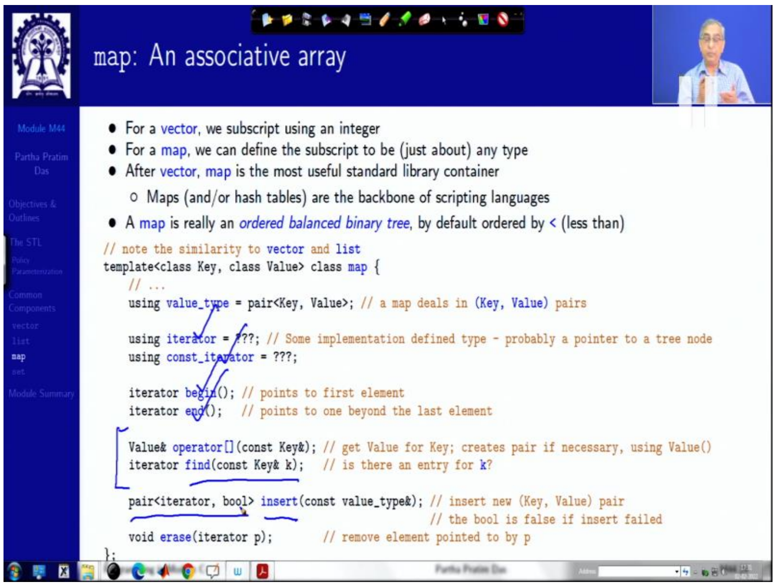
vector vs. list

- By default, use a vector
  - We need a reason not to
  - We can *grow* a vector (for example, using push_back())
  - We can insert() and erase() in a vector
  - Vector elements are *compactly stored* and *contiguous*
  - For small vectors of small elements all operations are fast
    - compared to lists
- If we do not want elements to move, use a list
  - We can *grow* a list (for example, using push_back() and push_front())
  - We can insert() and erase() in a list
  - List elements are *separately allocated*
- Note that there are more containers like
  - map
  - unordered_map [C++11]

Programming in Modern C++ Partha Pratim Das M44.28

But generically the code and the structure are all same and you can understand that on what context you should use a vector, obviously, if there is no other reason you should use a vector, it is a most efficient. You can grow both, vector and list. Vector can grow only at the back; list can go at both ends. You can do insert, delete. Naturally, vector has more compact storage contiguous, list have separate allocations.

(Refer Slide Time: 30:32)



map: An associative array

- For a vector, we subscript using an integer
- For a map, we can define the subscript to be (just about) any type
- After vector, map is the most useful standard library container
  - Maps (and/or hash tables) are the backbone of scripting languages
- A map is really an *ordered balanced binary tree*, by default ordered by < (less than)

```
// note the similarity to vector and list
template<class Key, class Value> class map {
    // ...
    using value_type = pair<Key, Value>; // a map deals in (Key, Value) pairs

    using iterator = ???; // Some implementation defined type - probably a pointer to a tree node
    using const_iterator = ???;

    iterator begin(); // points to first element
    iterator end();   // points to one beyond the last element

    Value& operator[](const Key&); // get Value for Key; creates pair if necessary, using Value()
    iterator find(const Key& k);   // is there an entry for k?

    pair<iterator, bool> insert(const value_type&); // insert new (Key, Value) pair
                                                    // the bool is false if insert failed
    void erase(iterator p);        // remove element pointed to by p
};
```

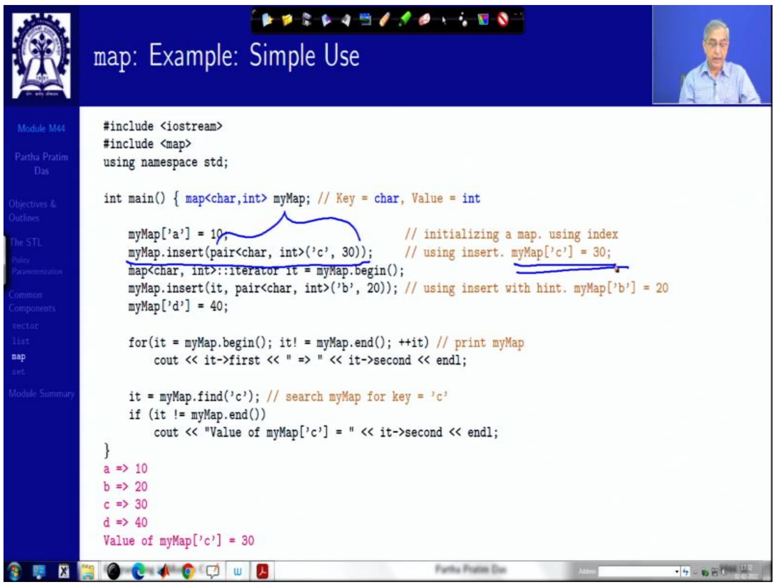
Let us look at the next container which is very very interesting, which is called a map. It is basically a kind of a, you know, it does not do hashing but it serves a similar purpose. In a vector, you use an integer as a subscript; in a map you can use anything as a subscript. Almost any type you can use, type of value you can use as a subscript.

After vector, map is probably the most useful standard library container to be used. It is implemented as an ordered balanced binary tree. So, let us look at the basic structure. I have the value type. So, map is a name value pair, you know. So, I say, this is ppd and this is his age, this pair always. So, the left one is called the key, by which you index. And the right one is the value that you….

So, the value_type for a map or the underlying types, element type for a map is a pair, a key type and the value type. And you pair them. Pair also is a, is available in the STL to take a pair of types and make it into a pair type which is becomes the value_type for the map. But you have the value type anyway. You have the iterator, constant iterator, begin, end, all these.

Additionally, in map, what you have? You have a way to access an element. Actually, you have this in vector also. You do not have it in list, because list cannot, is not indexed. You have a way to find, because in map finding is somewhat different. Your insert works differently, because now, you are not inserting based on the iterator, you insert based on a value and you get the basically the key value pair. Whereas erase works on the iterator itself.

(Refer Slide Time: 32:56)



map: Example: Simple Use

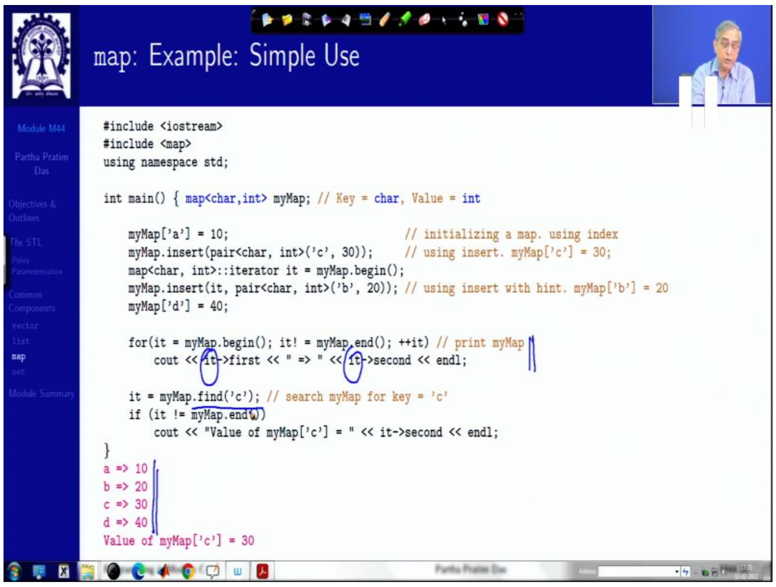
```
#include <iostream>
#include <map>
using namespace std;

int main() { map<char,int> myMap; // Key = char, Value = int

    myMap['a'] = 10;                          // initializing a map. using index
    myMap.insert(pair<char, int>('c', 30));   // using insert. myMap['c'] = 30;
    map<char, int>::iterator it = myMap.begin();
    myMap.insert(it, pair<char, int>('b', 20)); // using insert with hint. myMap['b'] = 20
    myMap['d'] = 40;

    for(it = myMap.begin(); it! = myMap.end(); ++it) // print myMap
        cout << it->first << " => " << it->second << endl;

    it = myMap.find('c'); // search myMap for key = 'c'
    if (it != myMap.end())
        cout << "Value of myMap['c'] = " << it->second << endl;
}
a => 10
b => 20
c => 30
d => 40
Value of myMap['c'] = 30
```

Let us just take a look at an example of a map. So, I am doing a map which is a char, int pair. So, key will be char value will be int. So, the pairs I am creating is ‘a’,10 that is character ‘a’,

10. So, you can see the standard array notation coming in here. I can just say my map, give that index value and assign the value. I can do this.

There are other ways of doing that also. I can do this by insert. If I want to do this by insert, I have to make a pair of these two values. Say, I am basically trying to do this. The key value is ‘c’, actual value is 30. So, I make a pair of them. I insert. So, these are different ways you can do you know insert, add elements to a map.

(Refer Slide Time: 34:00)

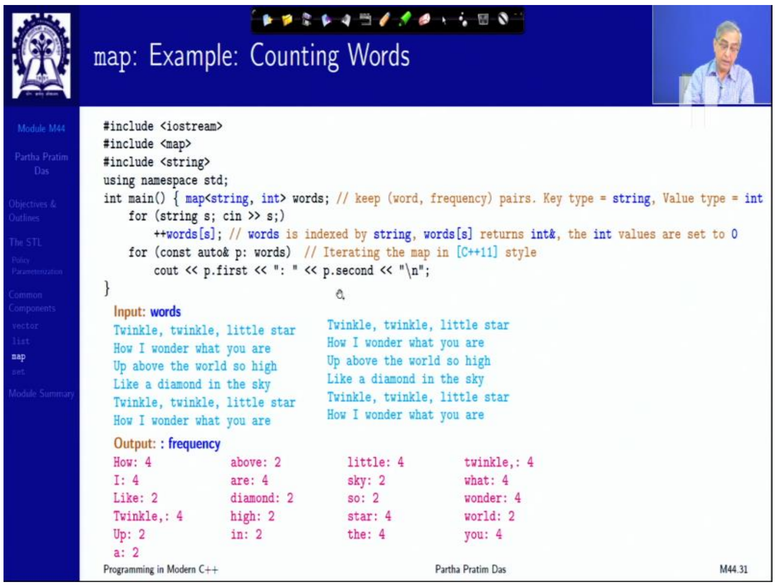


map: Example: Simple Use

```
#include <iostream>
#include <map>
using namespace std;

int main() { map<char,int> myMap; // Key = char, Value = int

    myMap['a'] = 10;                              // initializing a map. using index
    myMap.insert(pair<char, int>('c', 30));       // using insert. myMap['c'] = 30;
    map<char, int>::iterator it = myMap.begin();
    myMap.insert(it, pair<char, int>('b', 20)); // using insert with hint. myMap['b'] = 20
    myMap['d'] = 40;

    for(it = myMap.begin(); it! = myMap.end(); ++it) // print myMap
        cout << it->first << " => " << it->second << endl;

    it = myMap.find('c'); // search myMap for key = 'c'
    if (it != myMap.end())
        cout << "Value of myMap['c'] = " << it->second << endl;
}
a => 10
b => 20
c => 30
d => 40
Value of myMap['c'] = 30
```

Then you can do the printing using the basic iterator style here. Here you are getting the iterator and since map has two components of the pair, the first is the key second is the value, which prints these values. You can do a find on map. To given a key, you can find what is the value that it has. So, this is very very useful.

(Refer Slide Time: 34:28)



map: Example: Counting Words

Module M44

Partha Pratim Das

Objectives & Outlines

The STL

Policy Parameterization

Common Components

vector

list

map

set

Module Summary

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main() { map<string, int> words; // keep (word, frequency) pairs. Key type = string, Value type = int
    for (string s; cin >> s;)
        ++words[s]; // words is indexed by string, words[s] returns int&, the int values are set to 0
    for (const auto& p: words)  // Iterating the map in [C++11] style
        cout << p.first << ": " << p.second << "\n";
}
```

Input: words

```
Twinkle, twinkle, little star        Twinkle, twinkle, little star
How I wonder what you are            How I wonder what you are
Up above the world so high           Up above the world so high
Like a diamond in the sky            Like a diamond in the sky
Twinkle, twinkle, little star        Twinkle, twinkle, little star
How I wonder what you are            How I wonder what you are
```

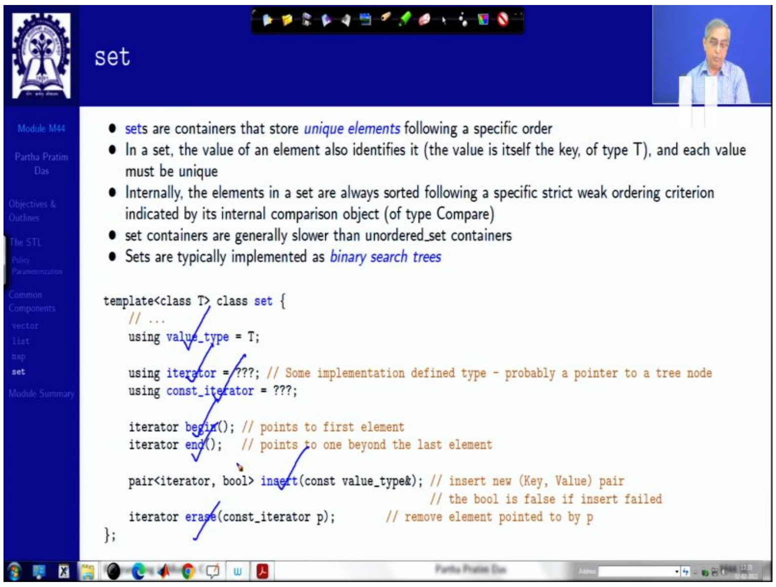
Output: : frequency

```
How: 4          above: 2        little: 4       twinkle,: 4
I: 4            are: 4          sky: 2          what: 4
Like: 2         diamond: 2      so: 2           wonder: 4
Twinkle,: 4     high: 2         star: 4         world: 2
Up: 2           in: 2           the: 4          you: 4
a: 2
```

Programming in Modern C++ Partha Pratim Das M44.31

And there is another example which is given here. I am not going through this. This is example of using a map to find the frequency of different words in a text. So, just try this out and convince yourself about the use of map.

(Refer Slide Time: 34:43)



set

Module M44

Partha Pratim Das

Objectives & Outlines

The STL

Policy Parameterization

Common Components

vector

list

map

set

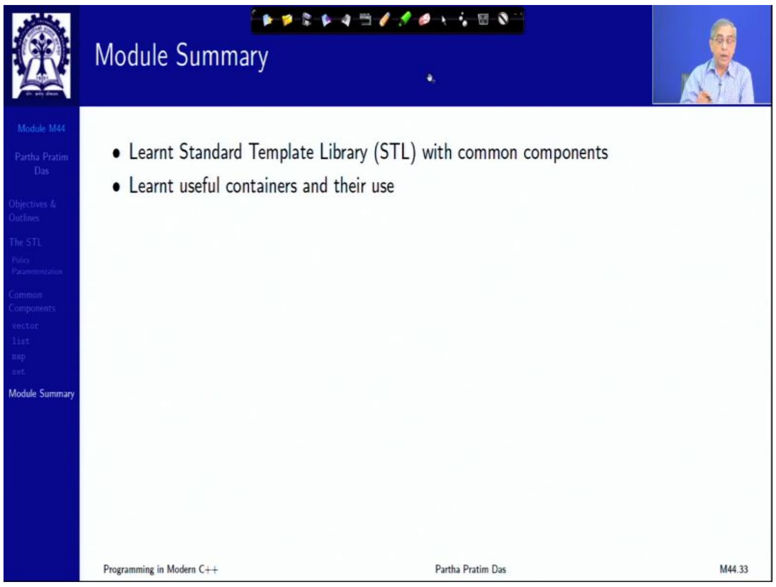
Module Summary

- sets are containers that store *unique elements* following a specific order
- In a set, the value of an element also identifies it (the value is itself the key, of type T), and each value must be unique
- Internally, the elements in a set are always sorted following a specific strict weak ordering criterion indicated by its internal comparison object (of type Compare)
- set containers are generally slower than unordered_set containers
- Sets are typically implemented as *binary search trees*

```
template<class T> class set {
    // ...
    using value_type = T;

    using iterator = ???; // Some implementation defined type - probably a pointer to a tree node
    using const_iterator = ???;

    iterator begin(); // points to first element
    iterator end();   // points to one beyond the last element

    pair<iterator, bool> insert(const value_type&); // insert new (Key, Value) pair
                                                  // the bool is false if insert failed
    iterator erase(const_iterator p);         // remove element pointed to by p
};
```

There are other containers also. Set is another which keeps this collection of unique elements. It is also stored as a binary search tree. And just wanted to show you the uniformity, the value type, iterator, concentrator, begin, end, insert, erase, just signatures are little bit here and there. Very similar to what you have for map, but set is also a useful container.

(Refer Slide Time: 35:12)



Module Summary

- Learnt Standard Template Library (STL) with common components
- Learnt useful containers and their use

There are many more, but it is time to close on this module. And here we have learnt about the standard template library with common components particularly, the focus has been on learning the basic containers, vector and map and some of the others that will also come in like string and set and so on. And what is their use. So, thank you very much for your attention and we will meet in the next module.