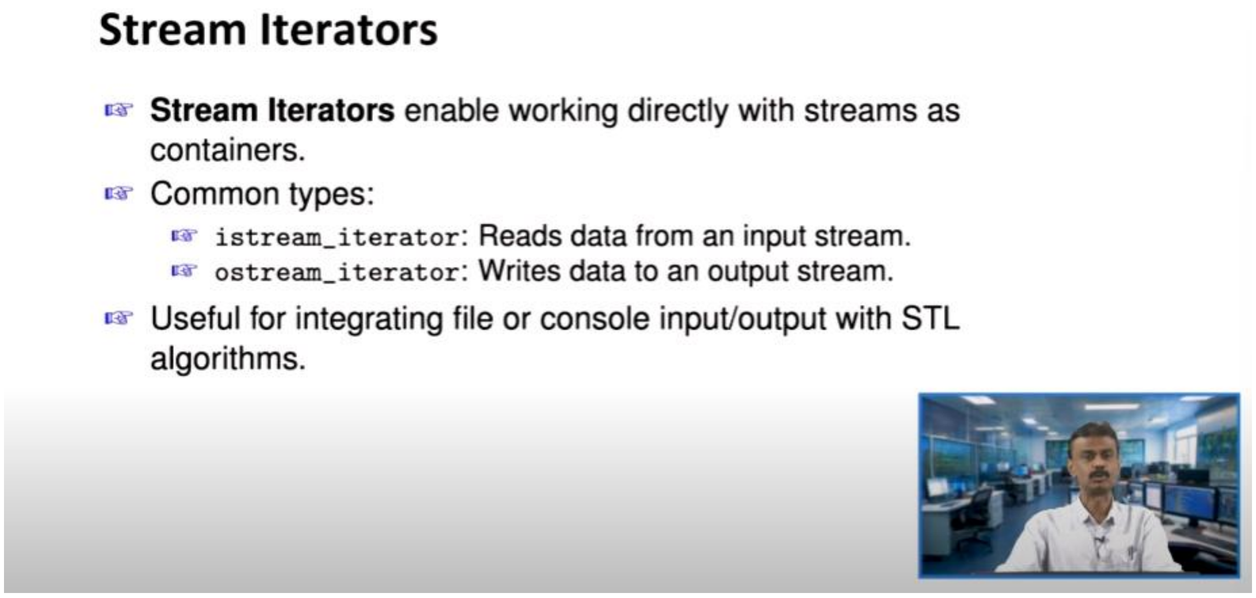
FUNDAMENTALS OF OBJECT ORIENTED PROGRAMMING

Lecture44

Lecture 44: STL Algorithms



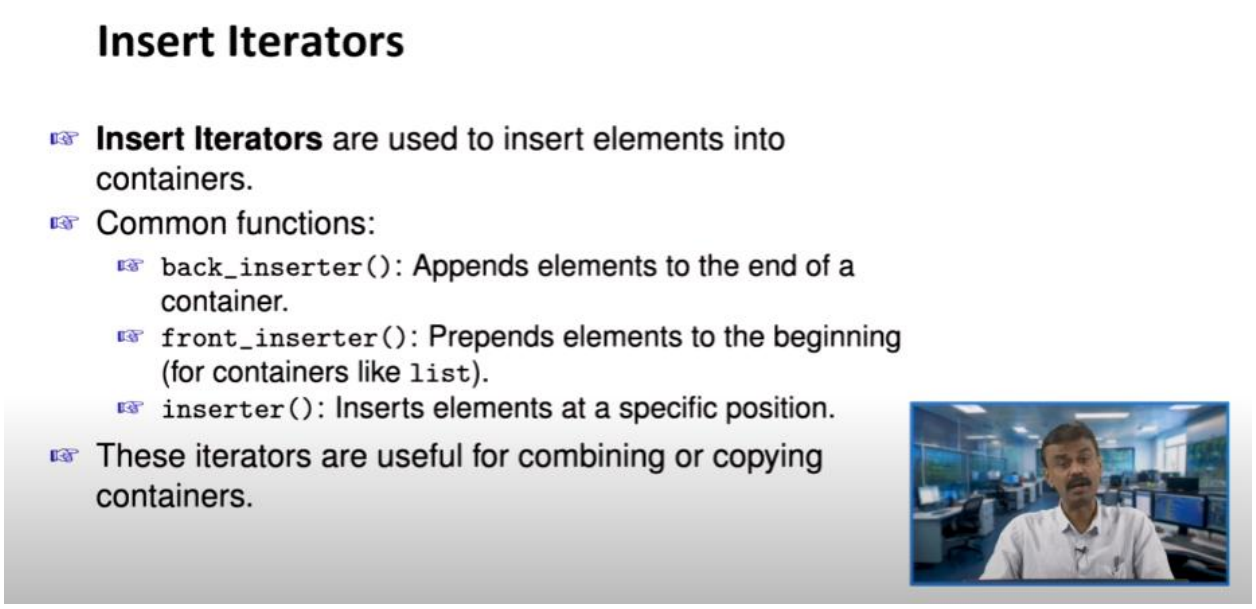
Stream Iterators

- **Stream Iterators** enable working directly with streams as containers.
- Common types:
  - `istream_iterator`: Reads data from an input stream.
  - `ostream_iterator`: Writes data to an output stream.
- Useful for integrating file or console input/output with STL algorithms.

Welcome to Lecture 44, the Standard Template Library. We call it STL. In today's lecture, I will start with stream iterators, right? So, stream iterators—I mean, you have common types like istream_iterator, right? That reads data from an input stream.

And similarly, for the output stream, you call it ostream_iterator. Which writes data to an output stream. So, these iterators—we call them stream iterators—they enable working directly with streams as containers, right? So, in the last lecture, we talked about containers. So here, I mean, when I am talking about stream iterators,

They enable working directly with streams as containers. So, as we have seen, istream_iterator means This particular function can read data from an input stream, right? And similarly, If I talk about ostream_iterator, that is nothing but the output stream, which writes data to an output stream. So, these are all useful.



Insert Iterators

- **Insert Iterators** are used to insert elements into containers.
- Common functions:
  - `back_inserter()`: Appends elements to the end of a container.
  - `front_inserter()`: Prepends elements to the beginning (for containers like `list`).
  - `inserter()`: Inserts elements at a specific position.
- These iterators are useful for combining or copying containers.

So, these are all the common types and these stream iterators which are useful for integrating file or when you are giving the input or getting the output in the console. So, this can be integrated with STL algorithm. So, you have a file or console input output. So, they can be integrated with STL algorithm. So, for this the stream iterators are very much useful.

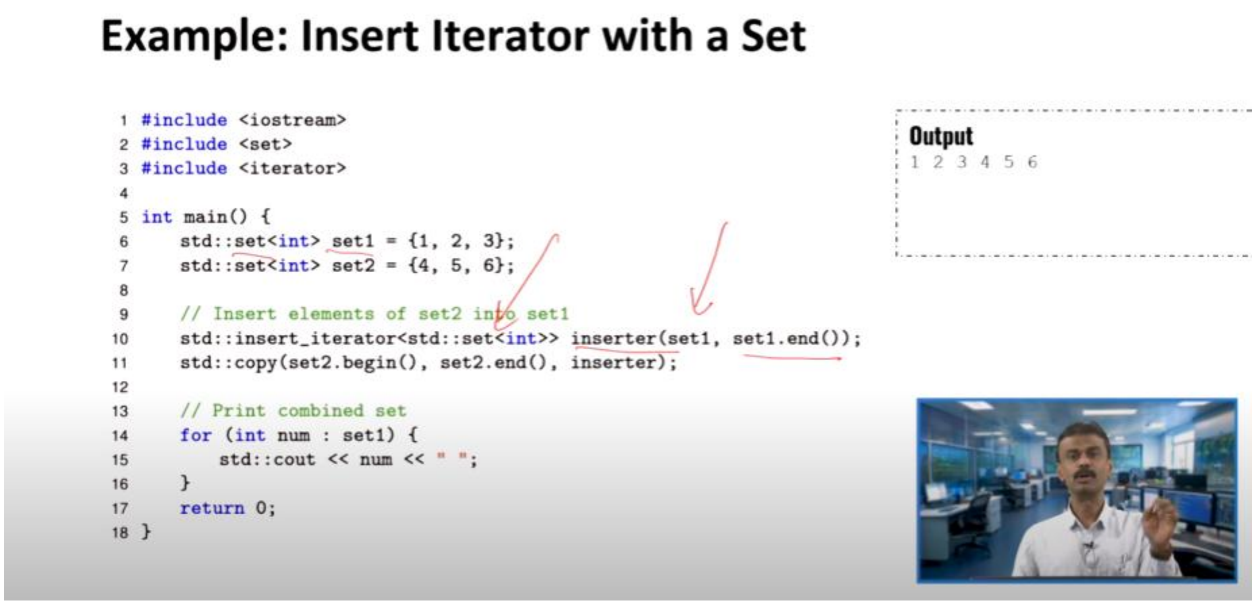
The next topic is insert iterators. Right. So, for example, I want to insert in the back. Right. So, you call it as a back underscore inserter.

These are all the common functions back underscore inserter and front underscore inserter. So, these are all used to insert elements into the containers. Right. So, similarly, you have inserter. Right.

One of the common functions is an inserter. Right. And I am talking about back_underscore inserter. So, they all append elements to the end of a container. It is exactly like in the last lecture we saw.

So, when you are inserting into the queue, either back or front. Exactly like this. And front inserter. That prepends elements to the beginning.

That means it puts the element at the beginning. And the common inserter. It is not a back or front. At any particular index, if you want to insert, or any particular position, you can insert the element. So, these are all the iterators.



Example: Insert Iterator with a Set

```
1  #include <iostream>
2  #include <set>
3  #include <iterator>
4
5  int main() {
6      std::set<int> set1 = {1, 2, 3};
7      std::set<int> set2 = {4, 5, 6};
8
9      // Insert elements of set2 into set1
10     std::insert_iterator<std::set<int>> inserter(set1, set1.end());
11     std::copy(set2.begin(), set2.end(), inserter);
12
13     // Print combined set
14     for (int num : set1) {
15         std::cout << num << " ";
16     }
17     return 0;
18 }
```

Output

```
1 2 3 4 5 6
```

So, they are all useful for combining or copying the containers, right? So, based on these examples, right? So, based on these concepts. So, let us see. I mean, we take an example of an insert iterator with a set, right?

So, here we include a set and an iterator. So, here we include a set and an iterator. If we look at line number 2 and line number 3 of this code. So, we take a set and an iterator. So, let us consider set1 under set.

A set of integers. So, one set we take 1, 2, 3, and another set we take 4, 5, and 6. So, now if you look at line number 10. So, you have a class insert_iterator. It is under iterator.

And this is the set. And set is nothing but the integer. Set is nothing but the set of integer. And you have an object inserter, right? So, you have an object inserter.

So, we are passing set1. So, set1 is an object under set, right? So, which are having value 1, 2, 3. So, set 1 if I put means it is nothing but you are having the base address. And another pointer set1 dot end, all right?

So, that means the whole set you are passing obviously the base address. So, that means it is nothing but set dot begin equivalent. And then you are having set one dot end. Right. So this means the base address and the end.

Right. So now, if you look at the next function. Right. So, you have copy. Right.

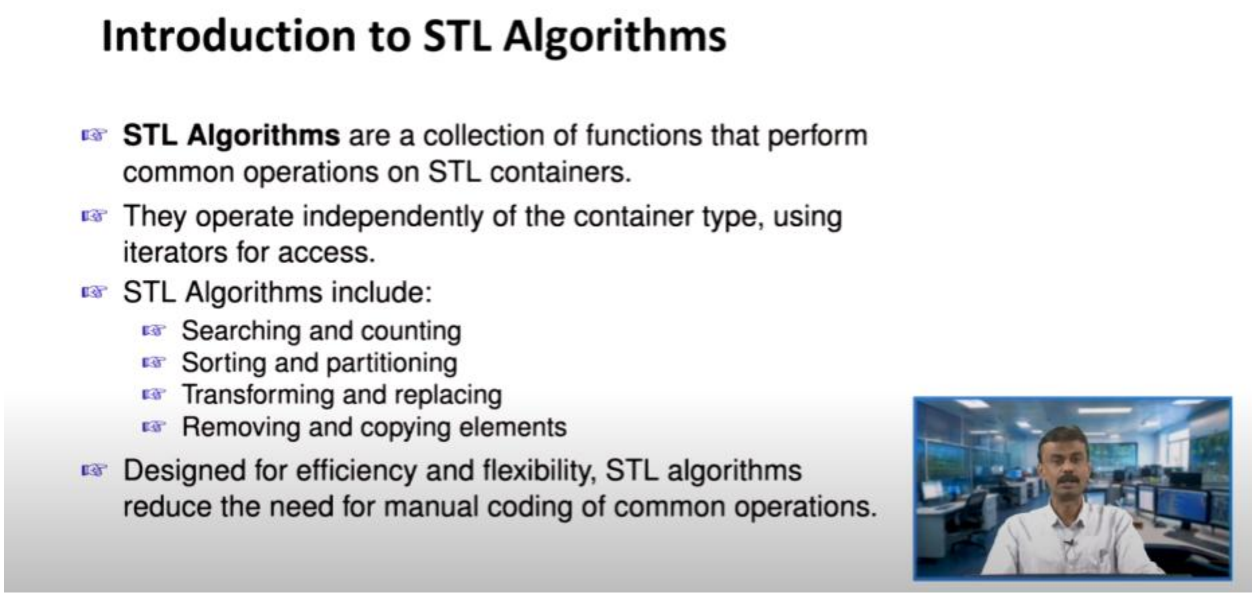
So here, it is an inserter, which is an object. So that means, by default. So these are all nothing but the inbuilt constructor. Whenever you are creating an object. So, this is a parameterized inbuilt constructor.

So, set1 is being passed, which means the base address of set1 and set1 dot end, as I put it in the arrow, right. So, set1 dot end pointer is also what you are passing. So now, if you look at line number 11, the copy function, you are calling the copy function. So, set2 dot begin, right, and set2 dot end, and here you go, the inserter, right, this particular object you are passing. So, this object contains set1 comma set1 dot end.

That means set1 dot begin and set1 dot end. So, copy. So, what it is being happen? So, whatever the content here, right? Whatever be the content here, set2 will be copied into set1, right?

So, the set2 content will be copied into set1. So, set1 you have 1, 2, 3 and set 2 you have 4, 5, 6. So, set2 will be copied into set1 that is the meaning of line number 11. So, the output we expect 1, 2, 3, 4, 5 and 6. So, now what I will do I will use this for loop set1 int num that means address of num1 and base address of set 1 pointing to the same all right.

So, now the number will take the index of the set1 array. As we said, set 1 should have, right, 6 elements and the 6 elements will be printed like this. So, what we can do? We can check this code in C++ compiler. So, if you run the code, right, so the same example that I have shown over here and if I execute and compile and run, you can see the output, right, 1, 2, 3, 4, 5, 6 as a output.



Introduction to STL Algorithms

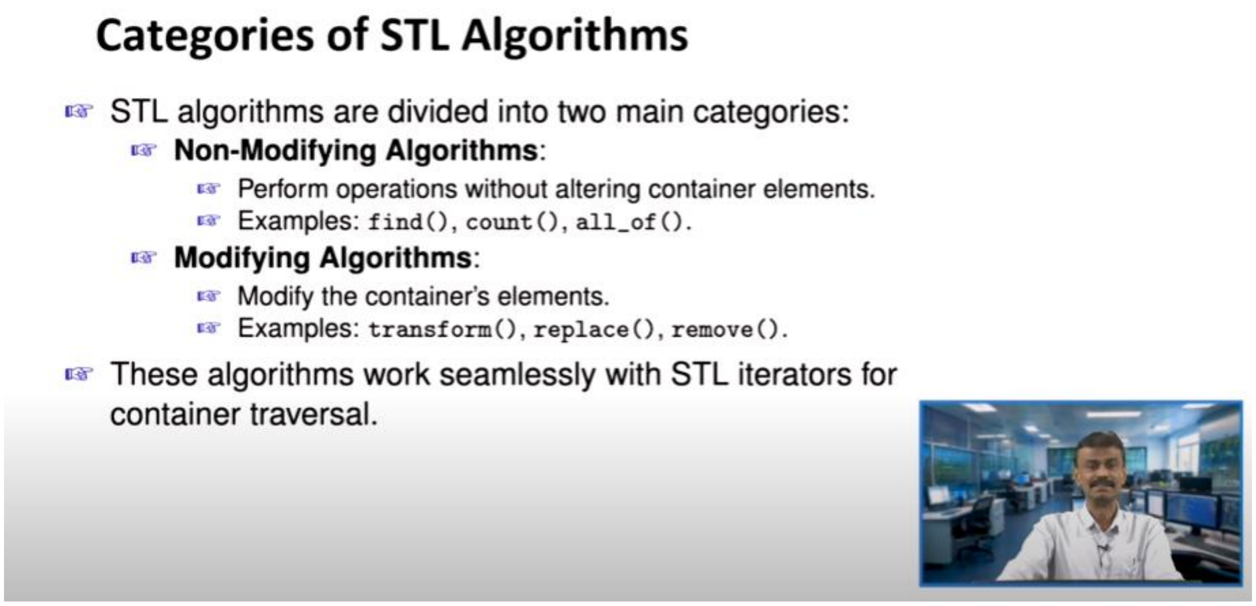
- **STL Algorithms** are a collection of functions that perform common operations on STL containers.
- They operate independently of the container type, using iterators for access.
- STL Algorithms include:
  - Searching and counting
  - Sorting and partitioning
  - Transforming and replacing
  - Removing and copying elements
- Designed for efficiency and flexibility, STL algorithms reduce the need for manual coding of common operations.

So, now, STL algorithms. So, in fact, when you are given a data structure, some of the data structures we have already seen. I talked about queues, stacks, and doubly linked lists. So, now, there are STL algorithms like searching, counting, and sorting. So, these are all nothing but collections of functions.

So, you are given an array or even a linked list. So, you have the inbuilt data. STL algorithms are nothing but collections of functions. So, these perform common operations on STL containers, and STL algorithms operate independently of the container type using iterators for access, right? Like whatever the iterators we have seen so far.

So, they use iterators for accessing. So, now, what are all the algorithms? STL algorithms: searching and counting, sorting and partitioning, transforming and replacing, removing and copying elements. So, there are several inbuilt algorithms. Right.

So, assume that you have not done any data structure course, right. So, it is like a black box. You can just call and in fact, some of the algorithms we are going to see, right. So, exclusively you will study in data structures, but those who are the beginners, assume that you have done your basic C, C++ program and you are extending this course, right, with object oriented. Assume that you do not know anything on the algorithm spot or data structure spot.



Categories of STL Algorithms

- STL algorithms are divided into two main categories:
  - **Non-Modifying Algorithms**:
    - Perform operations without altering container elements.
    - Examples: `find()`, `count()`, `all_of()`.
  - **Modifying Algorithms**:
    - Modify the container's elements.
    - Examples: `transform()`, `replace()`, `remove()`.
- These algorithms work seamlessly with STL iterators for container traversal.

So in that case, right, so you know what is going on. For example, if I say sorting means, so the random array will have, right, the output will be either ascending order or in the descending order. That is called sorting. Searching, I want to search an element in the array, counting the number of elements or number of occurrences of let us say 2s, the duplicates, right? Partitioning, partitioning into 2 arrays and we will see how we are going to do the transform, replace, removal of the elements or your copying into another array, right?

So, these are all the available algorithms, you can find it in STL. Right. So these are efficiently written. These inbuilt functions are efficiently written and they are also having the flexibility. Right.

So they have been designed. You have efficiency and flexibility. Right. And also, you know, need to write the manual coding. The main idea of this particular chapter, because you can see, in fact, I have introduced Python also.

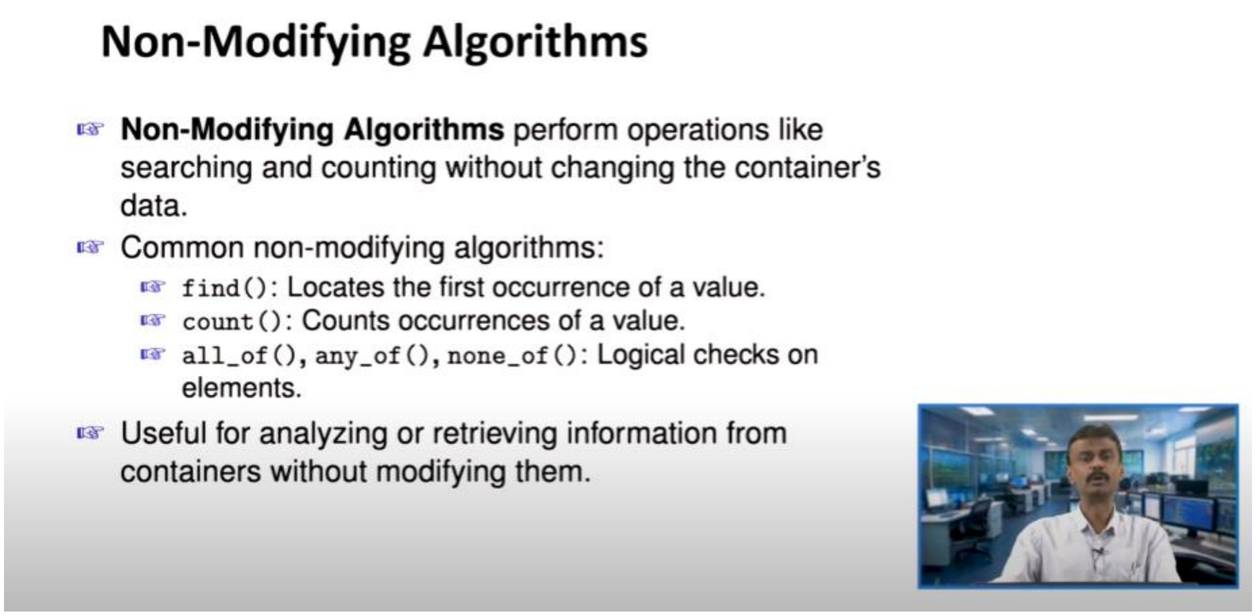
So everything based on the inbuilt functions. Yes, of course, you should know what is going on in the background, right. So, for that, you will separately study the courses like algorithms and data sets, right. So, here you have the common functions, right, the common inbuilt functions, correct. So, what you no need to do, you no need to write any manual code, right.

So, for example, if I call searching, it will do the search. If I do the counting, it will do the count, right. So, like this, we have several algorithms So, you call it as STL algorithm standard template library algorithms. So, how do you classify?

You can classify them as non-modifying algorithms and modifying algorithms. So, non-modifying algorithms have certain functions, right? I mean, you cannot do any changes in the container elements, right? For example, find, count are all examples of that, right? So, these are all the functions you call non-modifying algorithms.

There are several. We will see them, and similarly, modifying algorithms. So, you can do some small changes with the container elements, right? For example, if you are using the transform function, replace function, or remove functions, right? So, you can do a few modifications.

So, that is why these have been categorized as non-modifying algorithms and modifying algorithms, right? So, these algorithms work very efficiently and seamlessly with STL iterators for container traversal, okay? So, I will start with the non-modifying algorithms, right? As I already said, I mean, you cannot change the container elements or data, right? So, there are operations like find, right?



Non-Modifying Algorithms

- **Non-Modifying Algorithms** perform operations like searching and counting without changing the container's data.
- Common non-modifying algorithms:
  - `find()`: Locates the first occurrence of a value.
  - `count()`: Counts occurrences of a value.
  - `all_of()`, `any_of()`, `none_of()`: Logical checks on elements.
- Useful for analyzing or retrieving information from containers without modifying them.

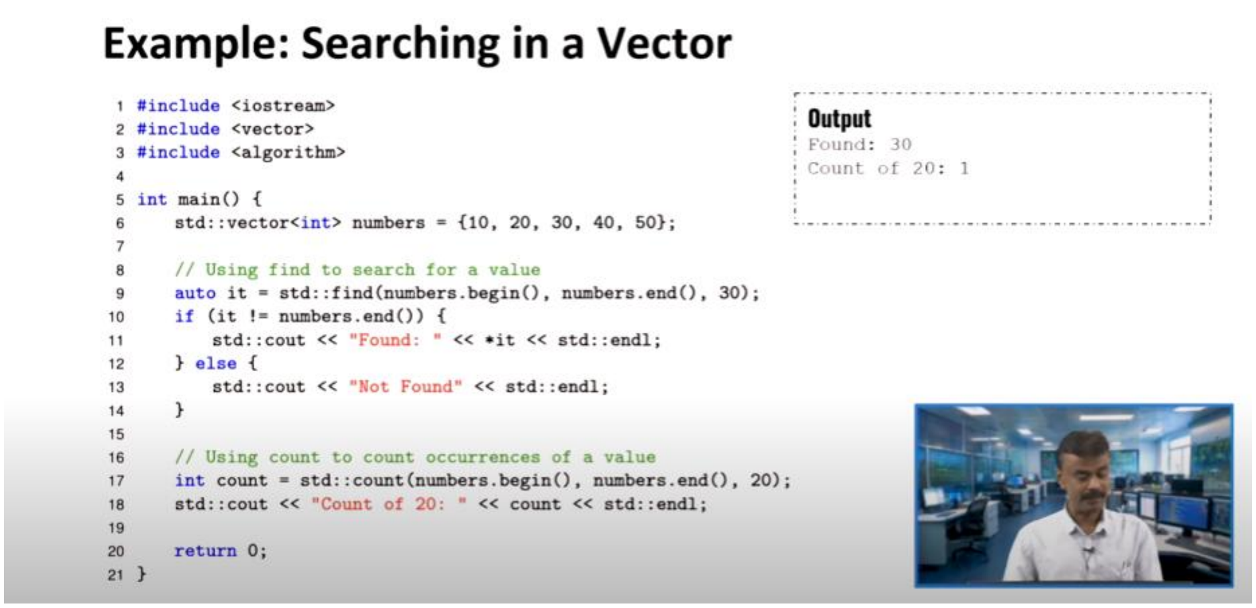
So, the find function, which is a non-modifying algorithm, locates the first occurrence of a value, right? Find. In an array of elements or a linked list of elements, I want to find element 2, right? Which node contains element 2 or which index contains node 2, right? So, you call it the find algorithm. Similarly, you have the count algorithm.

So, you can find the number of occurrences of, let us say, the value 20, right? You call the function count. Similarly, you have all_underscore_of, any_underscore_of, none_underscore_of. So, these are all nothing but logical checks on elements, right? So, logical checks.

So, these are all non-modifying algorithms. So, this is useful whenever you have applications where you do not need to modify anything. I want to find a given array. I want to find the element, right? I do not need to do any small code or modify the code, right? So, in those applications, For example, if you want to analyze or retrieve information, you can use non-modifying algorithms.

So, find, right. So, searching and counting in STL, find algorithm. As we have already seen, so the first occurrence, the given array or a linked list. So, what is the first match? So, I want to find number 10.

in a given array searching right. So, we know sequential search and binary search right. So, equivalent to that find you find out what is algorithm in fact you can find out right. So, you can see in the find function the non-modifying algorithm of STL what exact algorithm they have used right. So, this we can find out how they have developed correct.



Example: Searching in a Vector

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  int main() {
6      std::vector<int> numbers = {10, 20, 30, 40, 50};
7
8      // Using find to search for a value
9      auto it = std::find(numbers.begin(), numbers.end(), 30);
10     if (it != numbers.end()) {
11         std::cout << "Found: " << *it << std::endl;
12     } else {
13         std::cout << "Not Found" << std::endl;
14     }
15
16     // Using count to count occurrences of a value
17     int count = std::count(numbers.begin(), numbers.end(), 20);
18     std::cout << "Count of 20: " << count << std::endl;
19
20     return 0;
21 }
```

Output

```
Found: 30
Count of 20: 1
```

So, when I am calling this function find. So, it will search an element whichever is occurring first time and similarly count The number of occurrences of a specific value, how many times 20 present in the container, all right. So, these algorithms, so the find and count, you call it as a searching and counting. So, these algorithms work with any standard template library container using iterators.

So, let us consider this particular program: searching in a vector, all right. So, you have included vector and algorithm. First time we are including algorithm, all right, hash include algorithm. Line number 5, int main. You have the object numbers right under the class vector of integers, right? Vector of integers, like we call it as a set of integers right in the previous few examples.

So, vector of integers. So, which contains 10, 20, 30, 40, and 50, right. So, line number 9, you have auto variable right, it. Let us see what it is, right? Is auto variable. So, let us see.

Whether it is a pointer or an integer, we will come to know. So, you have an inbuilt function find. In fact, we had seen one or two slides back what find is doing. You are passing numbers dot begin and also you are passing numbers dot end. Meaning is, you are passing 10th position, 10th position, and 50th position, right?

And you want to search for an element 30. That is the meaning, right? When you are calling this function, it will return the location. Right. So this will return the location.

The find function will return the location. That means. Right. So this is the object, and it is a location. Correct.

It is like the numbers you have. What is the numbers dot begin? Right. It is equivalent to that. Right.

The location. So in that location, I want to find out. Right. I want to compare whether the searching element is found or not. Right.

So that is what we are going to do. If it not equal to numbers dot end right. So it is obviously right 30 position is 30 right. So now it will point over here that is the meaning. It is not pointing to 30 pointing to the location 30 right the address right.

It is equivalent to the reference where it is a object pointing to that particular location that is the meaning. So this is what we return. So now in line number 10 when you are comparing if it not equal to numbers dot end right. it not equal to numbers dot end. Yes, of course, this is end.

This is begin. True. If IT not equal to numbers dot end. End is this. Pointer to the location where the value 50 stored.

So, they are not equal. So, when they are not equal, it is true. IT not equal to numbers dot end is true. So, therefore, C out found will be printed. And then I am finding out what is the value star IT dereference.

right. So, it will print 30. So, found colon 30 will be printed. Yes, we are getting. We will check also, right.

So, the else part will not go. So, here you go one more, right. So, what we are doing one more. So, here we are using the count function. The count function will return the integer, right.

The count function will return the integer, right. So, numbers dot begin, correct. So, numbers dot begin. So, line number 9 we can consider it is the vector of int, right? You have the object like numbers. For example, when you want to print numbers, numbers will be like the base address, that means numbers dot begin, right.

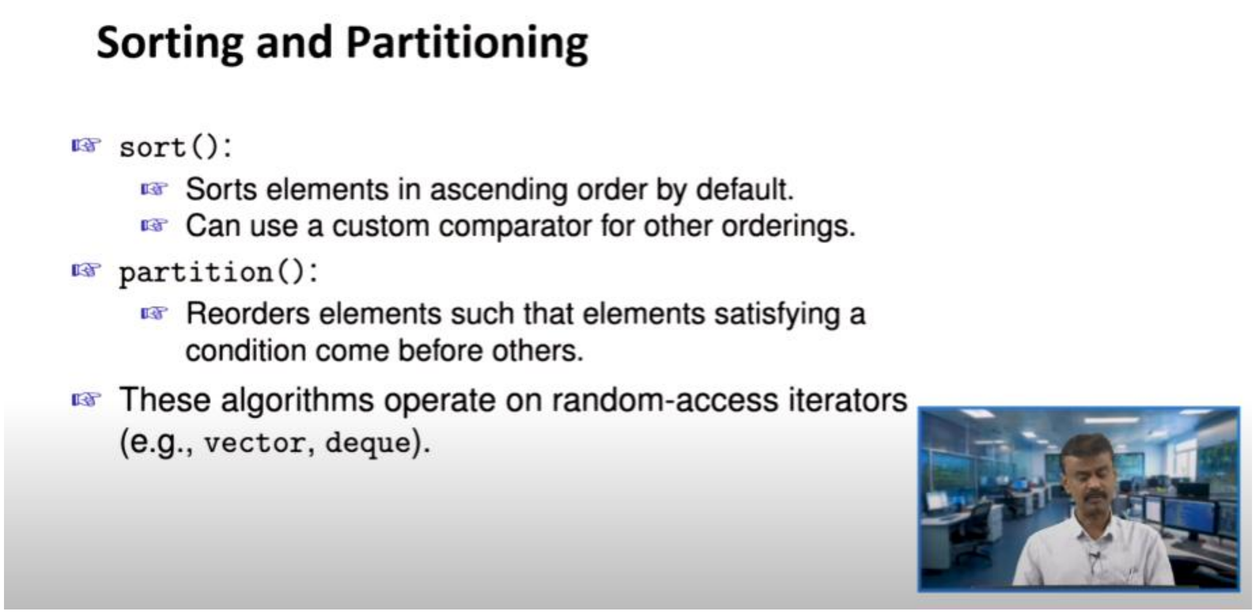
So here, in this case, line number 17, the return type is integer. So, numbers dot begin is being passed. Numbers dot end is being passed. You know which is begin and which is end. And here, I want to find the number of occurrences of 20.

Right. You are simply calling count. Right. And then whatever it returns. For example, it should return 1.

Because in the numbers vector of integers, we are finding only one instance of 20. Right. So, here we can expect the output to be 1. Right. So, that is what we are getting here.

Right. So, count of 20 colon. Count will be. So, 1 will be returned here. So, here we will have 1, right.

So, if we look at the output, found colon 30, and count of 20 is 1. So, we will run the code, right? So, here you go: numbers or objects, right? Vector of integers, and then you have—you can get found 20. Yes, 20 is found. If you look at the vector, vector of integers, and count of 20 is— So, 30 is found—found 30, count of 20 is 1, okay. So, similarly, we have sorting and partitioning.



Sorting and Partitioning

- `sort()`:
  - Sorts elements in ascending order by default.
  - Can use a custom comparator for other orderings.
- `partition()`:
  - Reorders elements such that elements satisfying a condition come before others.
- These algorithms operate on random-access iterators (e.g., `vector`, `deque`).

So, you can call the function sort, right? So, by default, it will sort in ascending order and partition. So, the partition will reorder the elements, right? Whereas the elements satisfying a condition come before others, right, right. For example, I want to take one element, and whichever element is less than this particular element should go in the left array, right—left side of an array.

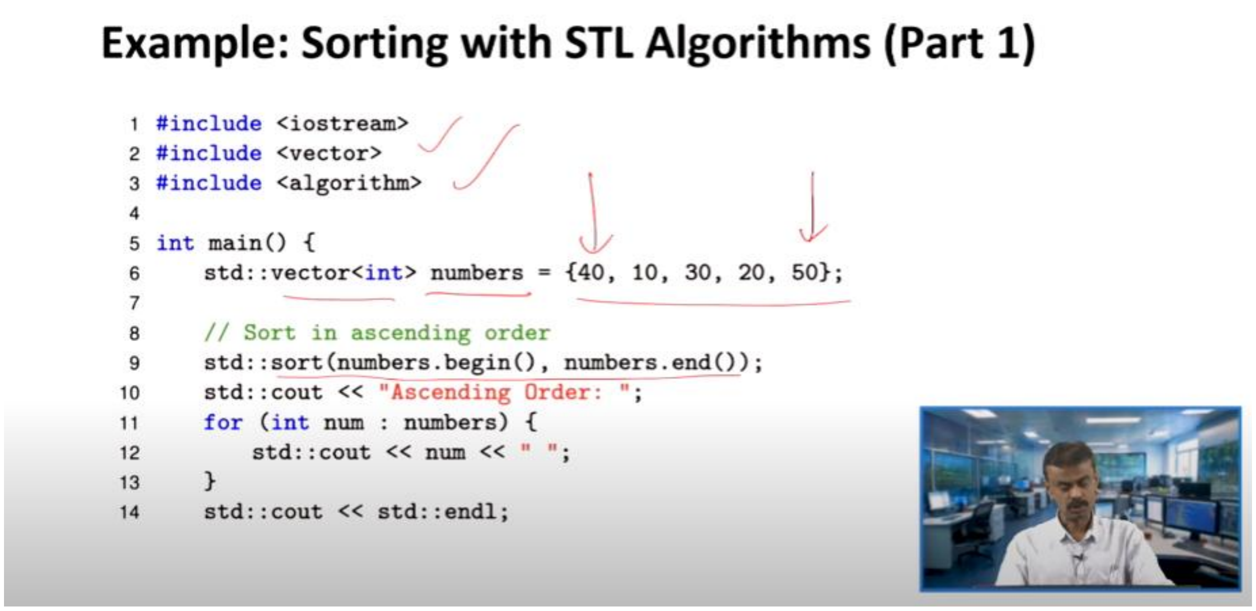
And whichever is greater than will go right side. So, this is a partition, right? So, certain condition—this is one example. So, partition—the function partition, right? So, these algorithms operate on random access iterators, vector and deck.

In fact, we had seen, right, vector and deque we had already seen. So, the random access iterators, the examples are vector and deque. So, these algorithms, sort and partition, operate on vectors and deques. So, let us consider this particular program: vector algorithm. I have a number of vectors of integers; these are the numbers, all right. So, here you go. I am calling the sort function, all right, sorting begin and end.

So, you are given 40 and 50. So, when you are calling the sort function, So, you will get them in ascending order, right. So, that means the first output I will get is 10, 20, 30, 40, and 50, right. And the program is continuing.

In fact, we are getting this. So, now we will see the case of descending order. So, here, for the first time, we are using a lambda function. So, the lambda function, you can see the notation like an array, right. And then you have the usual parameters that you are passing.

Right. So when here in the lambda function you are passing two values and when you have return a greater than b. Right. So whenever a is greater than b return 1. Right. Assume that 40 and 10 it will return 1.



Example: Sorting with STL Algorithms (Part 1)

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers = {40, 10, 30, 20, 50};

    // Sort in ascending order
    std::sort(numbers.begin(), numbers.end());
    std::cout << "Ascending Order: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
```

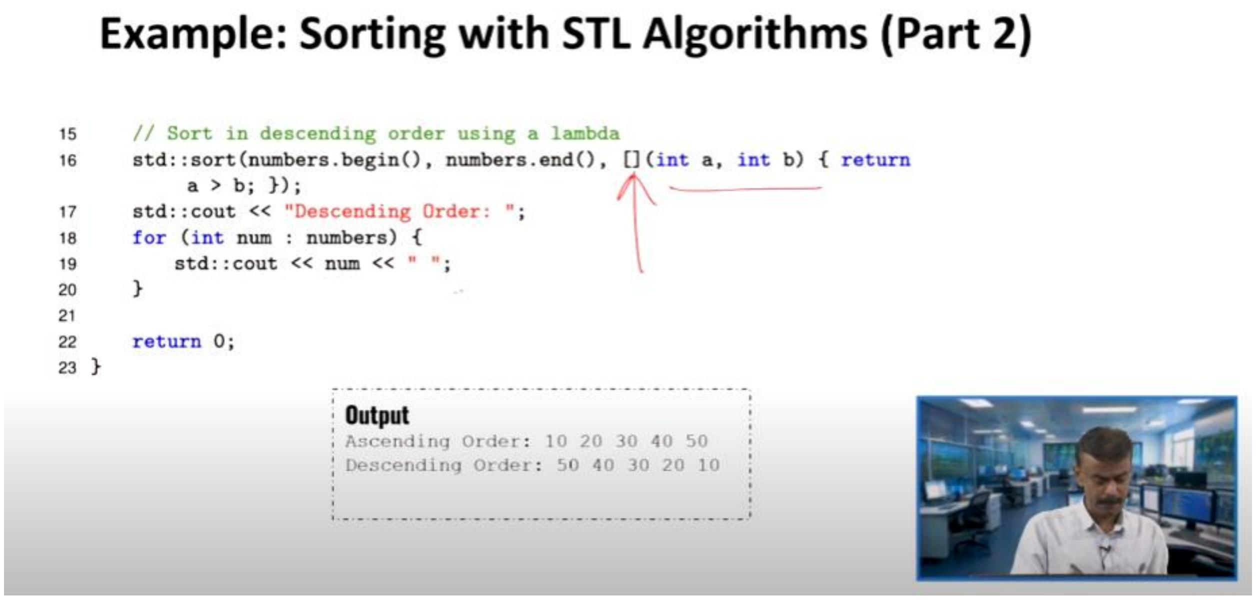
Right. So in the previous case it is as such assume that if you are using a lambda function in line number 9. Right. How do you use the logic? The lambda function int a comma int b return a less than b. Right.

So, that is a usual one, but you no need to write because inbuilt, it will be sorting in the ascending order, right. So, here in this case, it will be in the descending order, right, A greater than B, all right. So, then your all your numbers will be in descending order, that will be mapped onto number. When you want to print out, right, when you want to print, you are getting the descending order, right. Descending order will be printed, 50, 40, 30, 20, 10 will be printed.

So, this we can sort using the lambda function, right? So, this is the first time we are using the lambda function. So, these are all the non-modifying algorithms. So, now we will talk about the modifying algorithms. So, the modifying algorithms you have are transform, replace, and remove, right?

So, these are all the modifying algorithms. So, you can do some alterations, right? I mean, you can alter the elements of a container. So, for example, transform applies a function to each element and stores the result. Replace replaces all occurrences of a value with another value.

So, suppose 20 is occurring in the array more than, let us say, one time, and you want to change it to 30, right? So, you call the replace function. Remove lets you remove the elements, right? Suppose I want to remove 10; you can do that, right? So, these are all things you can do logically. So, these algorithms—the modifying algorithms—work with iterators to directly modify container data, all right?



Example: Sorting with STL Algorithms (Part 2)

```
    // Sort in descending order using a lambda
    std::sort(numbers.begin(), numbers.end(), [](int a, int b) { return
        a > b; });
    std::cout << "Descending Order: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }

    return 0;
}
```

Output

```
Ascending Order: 10 20 30 40 50
Descending Order: 50 40 30 20 10
```

So, you can do the modification, whereas in the previous case, either in the ascending or descending order, the same data were there, right? So, here you can do the modification, right? So, these are all the transform and replace algorithms, alright? So, as I already said. So, the transform function.

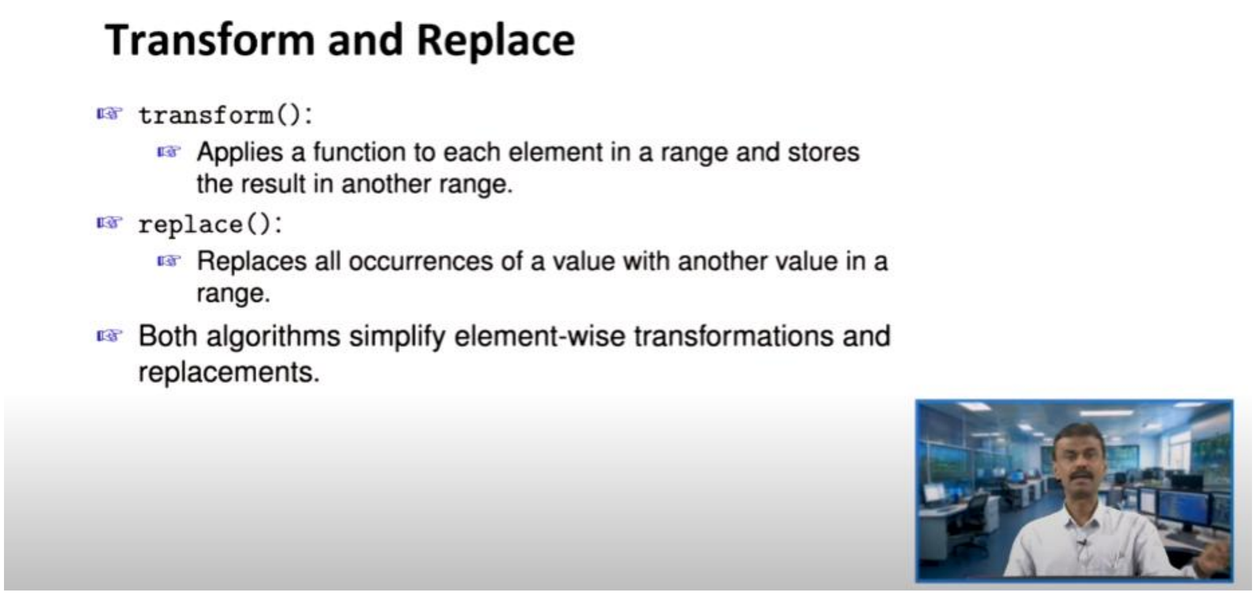
So, you can apply to each element, and then the result is correct. It is like Fourier transform: your integer is the input—assume that, right?—integer is the input, and a complex number you are getting as output, right? So, in a similar way. So, you apply, alright? An integer may change to a real value, right? After applying some mathematical transformation, it may change to a complex value, correct? So, this is a transform—that is what exactly it is written.

Apply to each element, and then whatever be the resultant, the resultant may be another range. Replace all occurrences of a value with another value. So, that means you can do the modification, right, with the data. So, both these algorithms. So, element-wise you are applying, and then you are changing the values.

So, let us consider transform all right. We take an example of a transform. So, for example, yeah, vector and algorithm have been included. You have numbers 1, 2, 3, 4, 5, right? And then I have another object called squared.

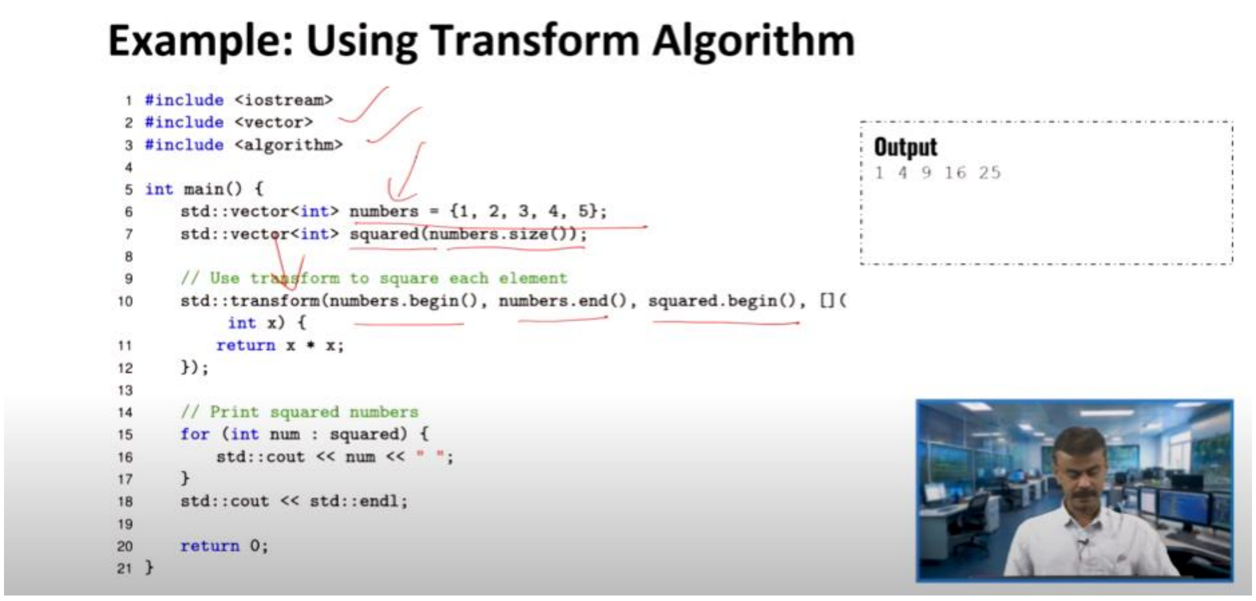
Numbers is a, as usual, object that contains 1, 2, 3, 4, 5. Another object squared. So, size is same as number. That is the meaning. And then it will be initialized.

All the values will be initialized to 0, all right? So, now I call transform function, all right? So here you go, numbers begin, numbers end and squared begin, right? The first pointer, right? The base address of the squared object, right?



Transform and Replace

- transform():
  - Applies a function to each element in a range and stores the result in another range.
- replace():
  - Replaces all occurrences of a value with another value in a range.
- Both algorithms simplify element-wise transformations and replacements.



Example: Using Transform Algorithm

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    std::vector<int> squared(numbers.size());

    // Use transform to square each element
    std::transform(numbers.begin(), numbers.end(), squared.begin(), [](
        int x) {
        return x * x;
    });

    // Print squared numbers
    for (int num : squared) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Output

```
1 4 9 16 25
```

I am again using the lambda function. You have an argument x, and this is returning x star x, right? So, this is returning x star x. That means numbers begin, numbers end. 1, 2, 3, 4, 5 you are entering. So, that will be acting as an x, right?

It is built-in. So, the resultant 1 into 1, the 1 will be stored in the base address of this squared. 2 into 2, 4 will be stored in the next location. 3 into 3 will be stored. So, when I print this, we will get this as the output.

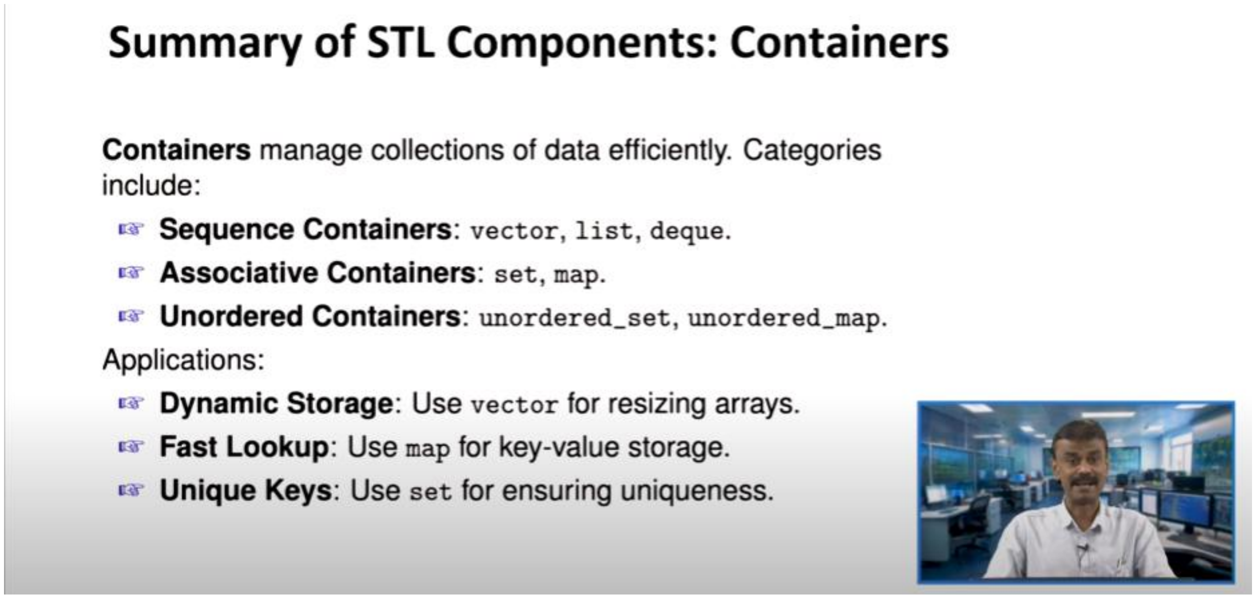
Now, I have completely modified the data. The original data is 1, 2, 3, 4, 5, and you can see the output is 1, 4, 9, 16, 25. So, the transform is modifying the data, right? So, now, the summary of STL components containers, right? So, in fact, whatever we have seen, I am going to summarize, right?

So, we talked about containers, in fact. We know what you mean by sequence containers, associative containers, unordered containers, right? Like sequence containers, if you take vector, list, deque—we studied all. Right? Associative containers, we studied set and map. Unordered set and unordered map we have done in the previous lectures, right? So, they are all nothing but unordered containers. And we had seen several applications. We have used vector for resizing arrays; you call it as a dynamic storage. And we have used the key, right? The key-value storage with the help of map.

You call it as a lookup table or fast lookup. And we use the set for the uniqueness of the keys. You call it as unique keys. And similarly, we dealt with the iterators, right? Input iterator reads the element in forward direction.

Output iterator writes the element in forward direction. Forward iterator reads and writes sequentially. Right? And then we had a bidirectional, right? Moves both forward as well as backward—random access. So, you can go to some particular position right in constant time. Suppose you use vector and deque—yes, you can do that. This is also we have studied: random access iterators. So, the applications are sequential traversal.

So, the vector and list we can use the sequential algorithm for traversal and reverse iteration. Right, reverse begin and reverse end, if you recall this, we have used an algorithm integration just now. We talked about transform, or sort. All right, so these are all the summaries of STL components. For one, we had seen iterators; another one, we had seen containers. So, in today's lecture, we talked about STL algorithms, right? Non-modifying algorithms and modifying algorithms. If you use the functions like find, count, all underscore of, we studied, they are all non-modifying algorithms. And in the case of modifying algorithms, we particularly worked with transform. One of the examples we had seen with the help of transform. So, transform, replace, remove—they are all considered as modifying algorithms, right? So, these are all examples. So, we had seen some of them, and the main applications are searching and sorting. So, we use find and sort functions, correct?



Summary of STL Components: Containers

Containers manage collections of data efficiently. Categories include:

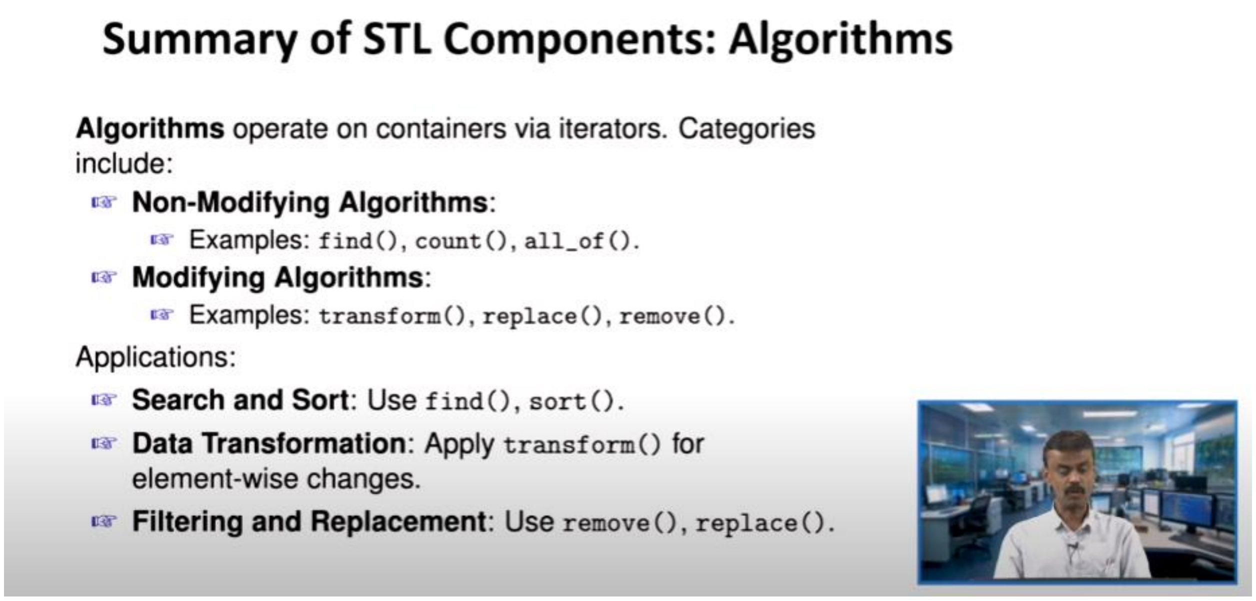
- **Sequence Containers**: `vector`, `list`, `deque`.
- **Associative Containers**: `set`, `map`.
- **Unordered Containers**: `unordered_set`, `unordered_map`.

Applications:

- **Dynamic Storage**: Use `vector` for resizing arrays.
- **Fast Lookup**: Use `map` for key-value storage.
- **Unique Keys**: Use `set` for ensuring uniqueness.

So, the find is non-modifying, right? And another one is the modified, like transform, right? Data transformation from the element-wise if you apply, it is changing to another one, and filtering and replacement, we use remove and replace, right? So maybe you work it out. What exactly is the remove doing?

What exactly is the replace doing? Right. So these are all. In fact, I mean, we have seen how find and sort work. Correct.



Summary of STL Components: Algorithms

Algorithms operate on containers via iterators. Categories include:

- **Non-Modifying Algorithms**:
  - Examples: `find()`, `count()`, `all_of()`.
- **Modifying Algorithms**:
  - Examples: `transform()`, `replace()`, `remove()`.

Applications:

- **Search and Sort**: Use `find()`, `sort()`.
- **Data Transformation**: Apply `transform()` for element-wise changes.
- **Filtering and Replacement**: Use `remove()`, `replace()`.

So we have done. All right. And we have also done the transform. So now you have to take it as an assignment. How does remove work?

How does replace work? You have to extend whatever we have seen. I mean, assume that you are given an array of elements or a vector of elements: 10, 20, 30, 40, 50. And the task is to remove 30. Right.

So, you can do that and similarly replace it. So, these are all the algorithms you can practice. So, with this, I am completing all these concepts, right? In the next lecture, what we can do is see some case studies, right? Then, I can conclude this particular chapter. Thank you.