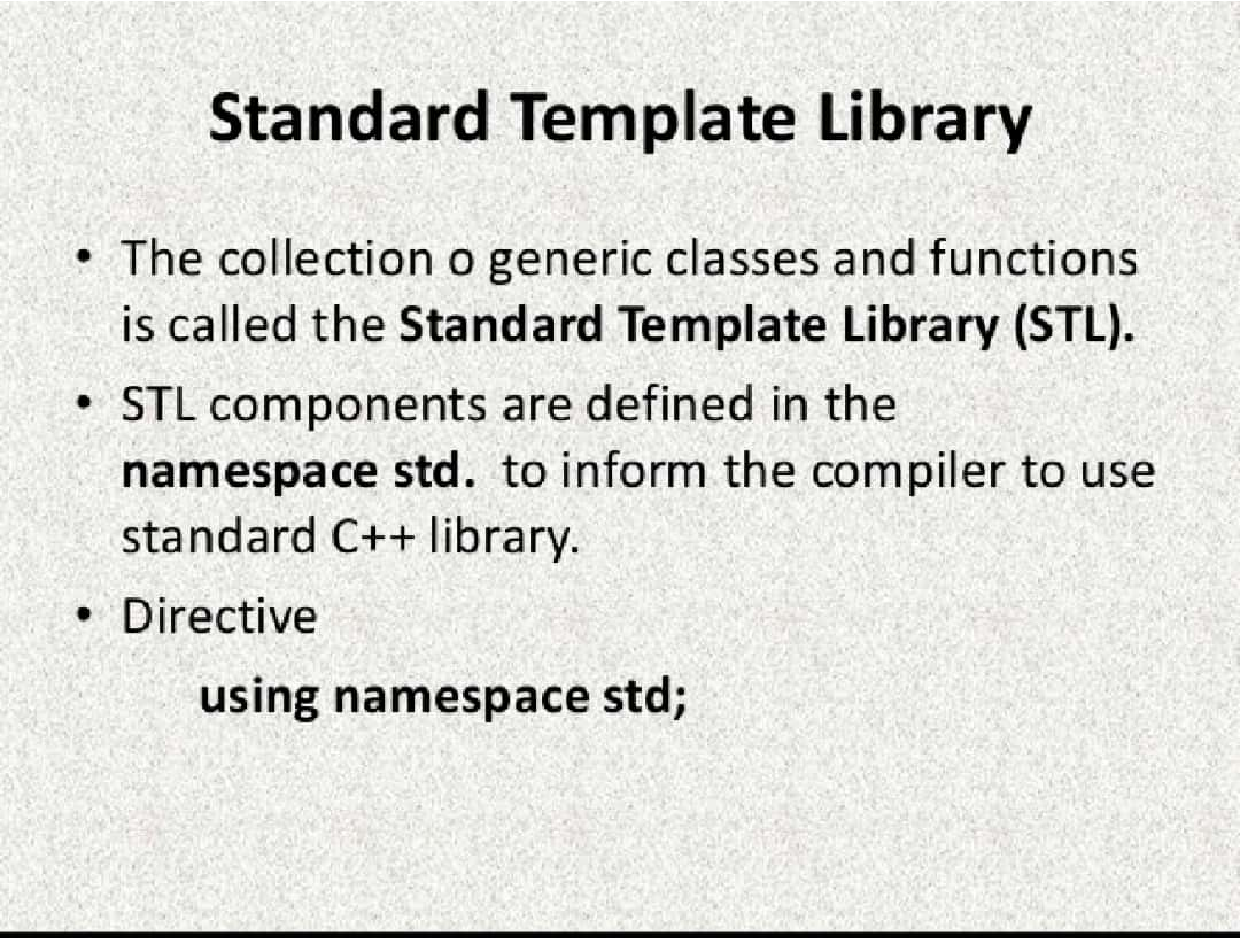
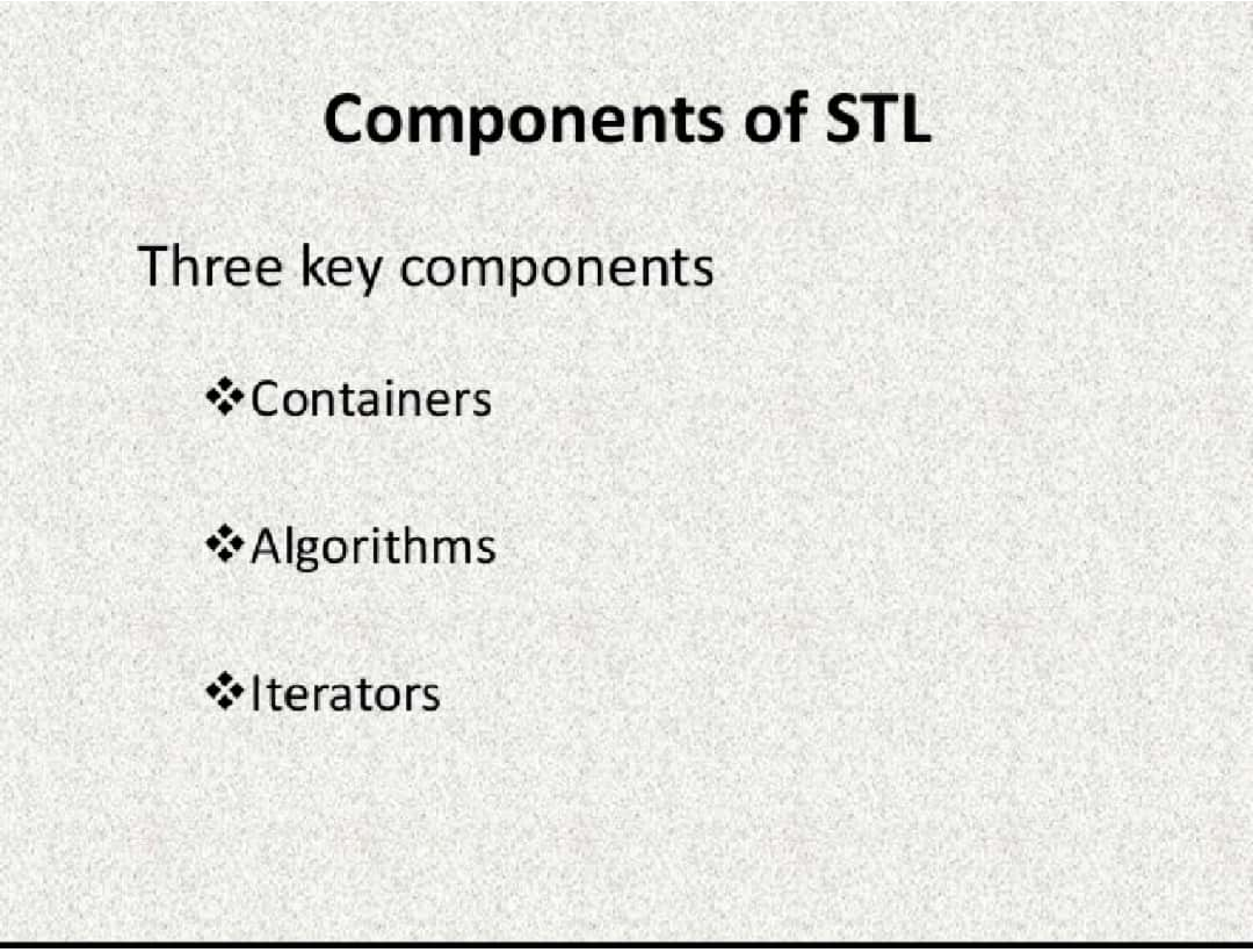


Standard Template Library

- The collection o generic classes and functions is called the **Standard Template Library (STL).**
- STL components are defined in the **namespace std.** to inform the compiler to use standard C++ library.
- Directive

  using namespace std;

Scanned by CamScanner



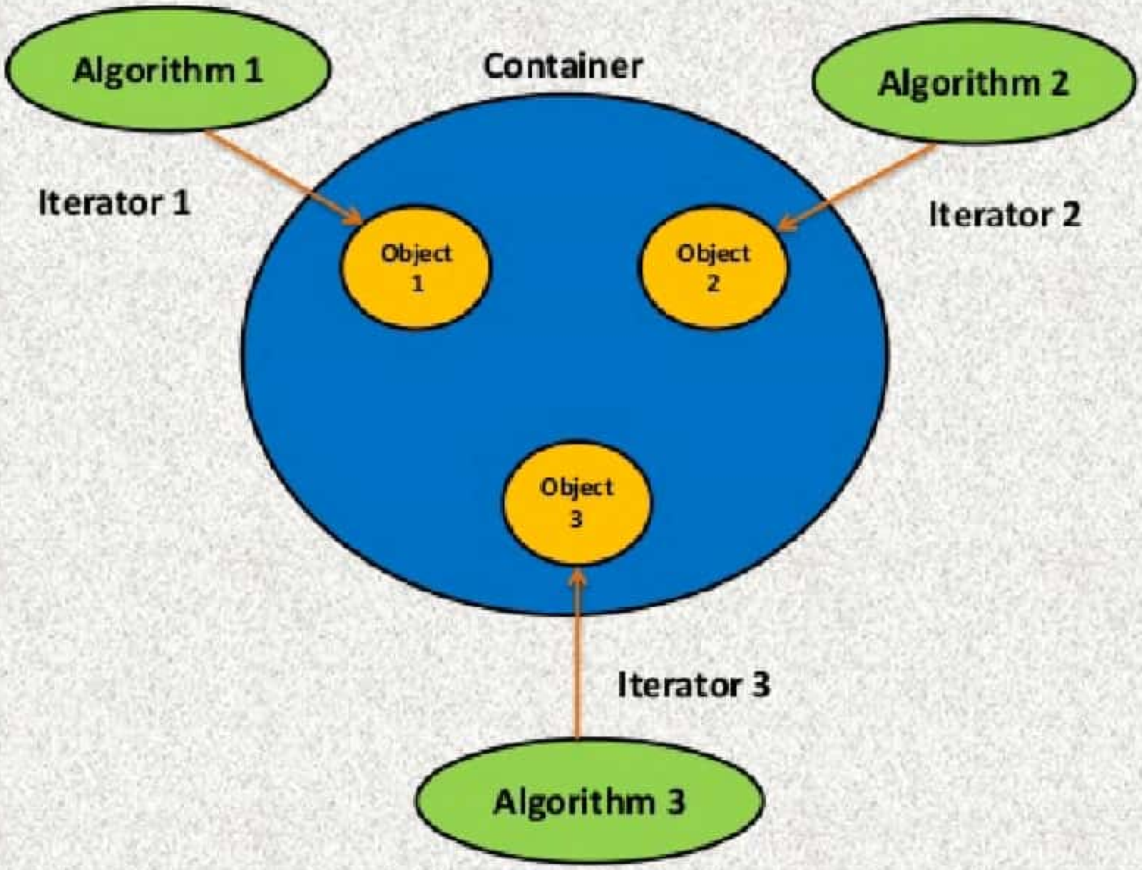
Components of STL

Three key components

- ❖Containers
- ❖Algorithms
- ❖Iterators

Scanned by CamScanner

Relationship between Three STL Components



Algorithm 1

Container

Algorithm 2

Iterator 1

Iterator 2

Object 1

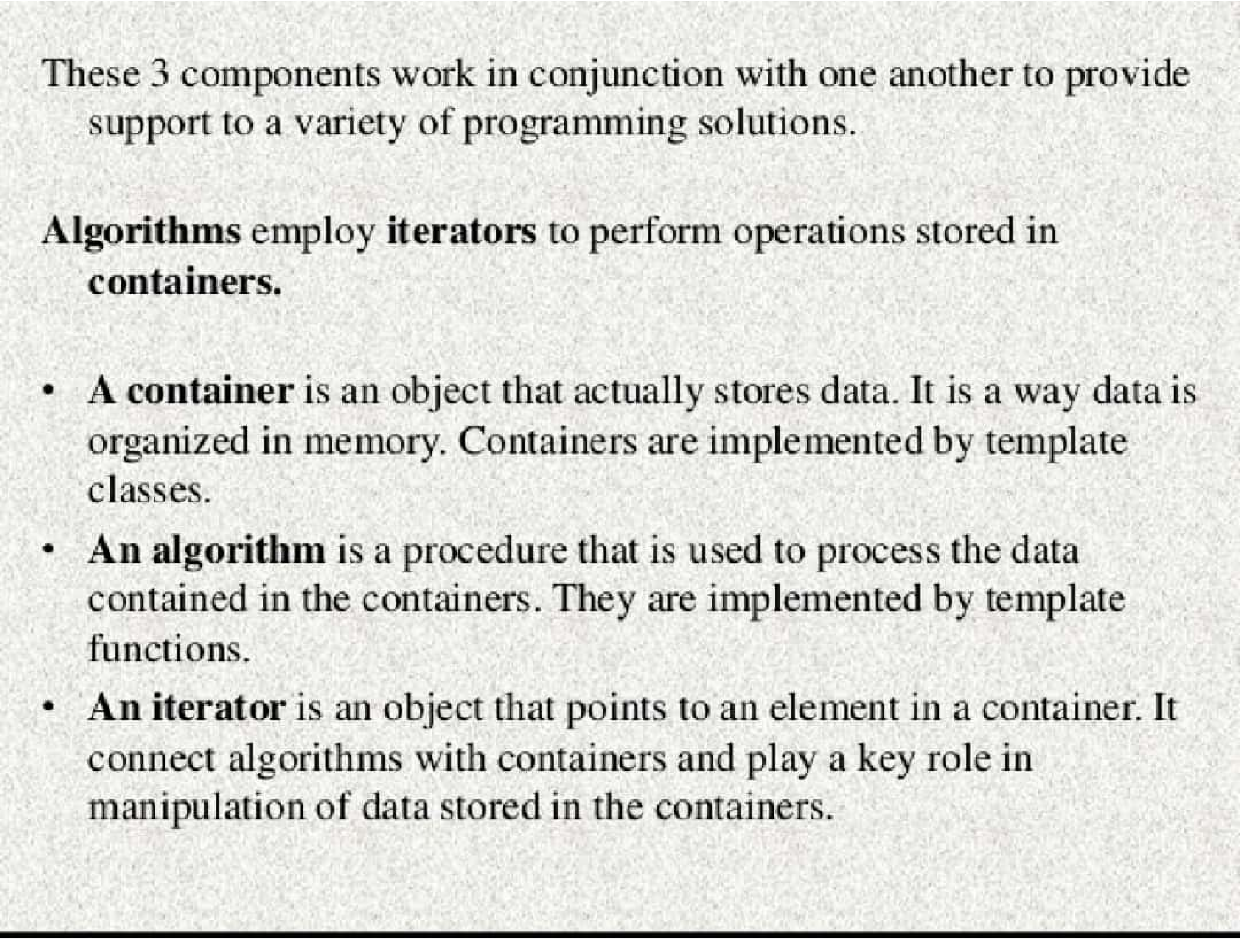
Object 2

Object 3

Iterator 3

Algorithm 3

Scanned by CamScanner

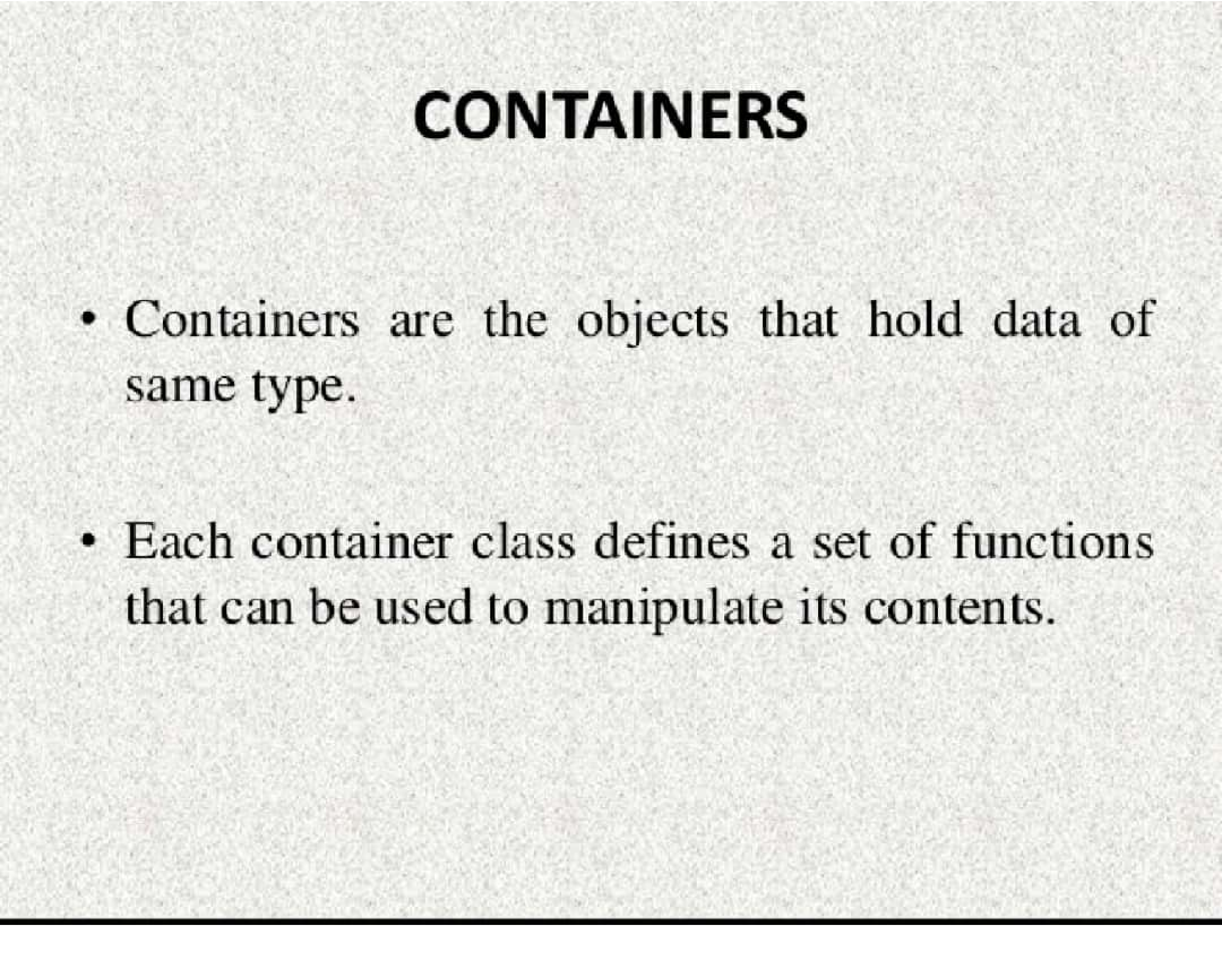


These 3 components work in conjunction with one another to provide support to a variety of programming solutions.

Algorithms employ **iterators** to perform operations stored in **containers.**

- **A container** is an object that actually stores data. It is a way data is organized in memory. Containers are implemented by template classes.
- **An algorithm** is a procedure that is used to process the data contained in the containers. They are implemented by template functions.
- **An iterator** is an object that points to an element in a container. It connect algorithms with containers and play a key role in manipulation of data stored in the containers.

Scanned by CamScanner

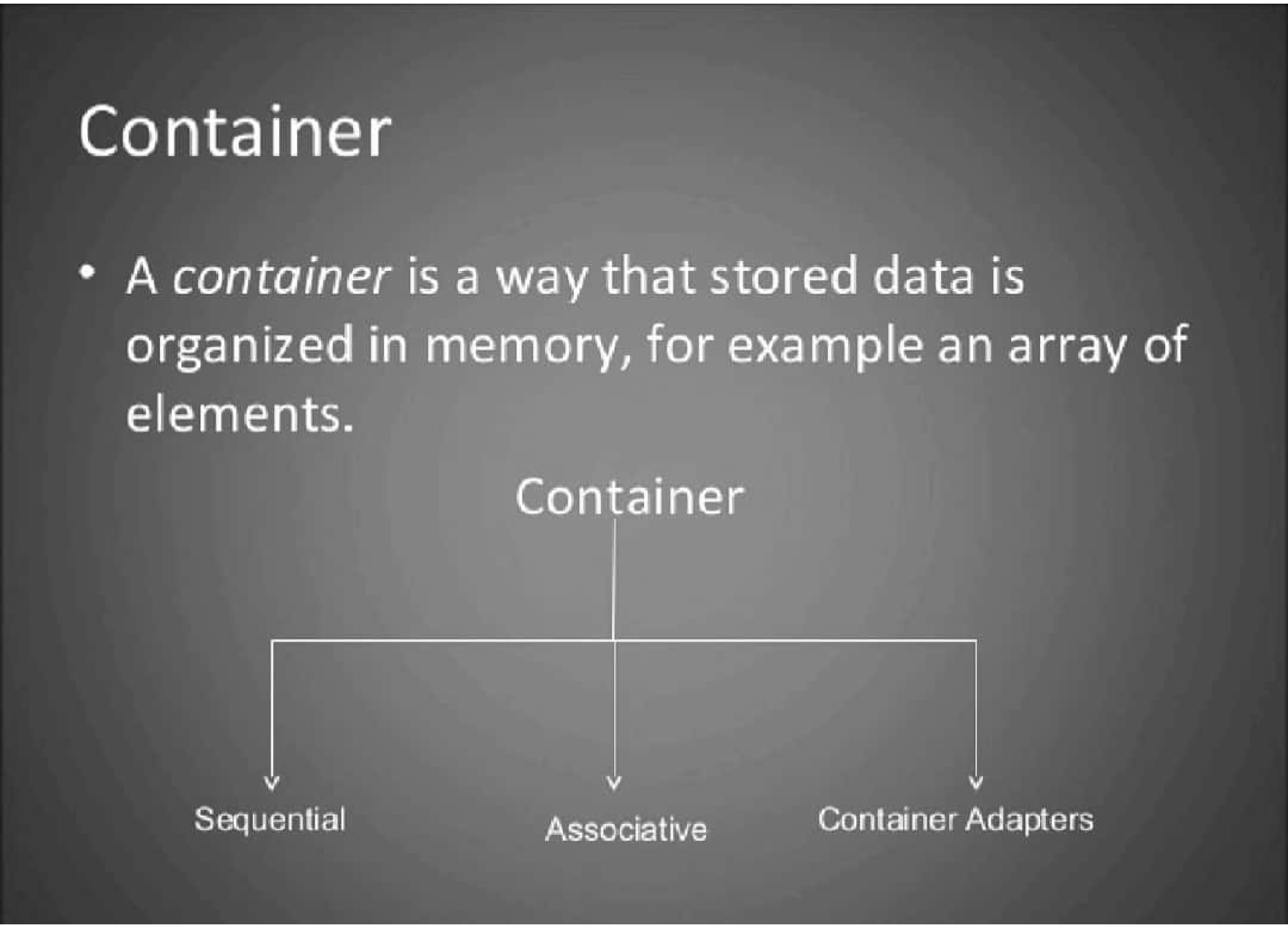


CONTAINERS

- Containers are the objects that hold data of same type.

- Each container class defines a set of functions that can be used to manipulate its contents.

Scanned by CamScanner



Container

- A *container* is a way that stored data is organized in memory, for example an array of elements.

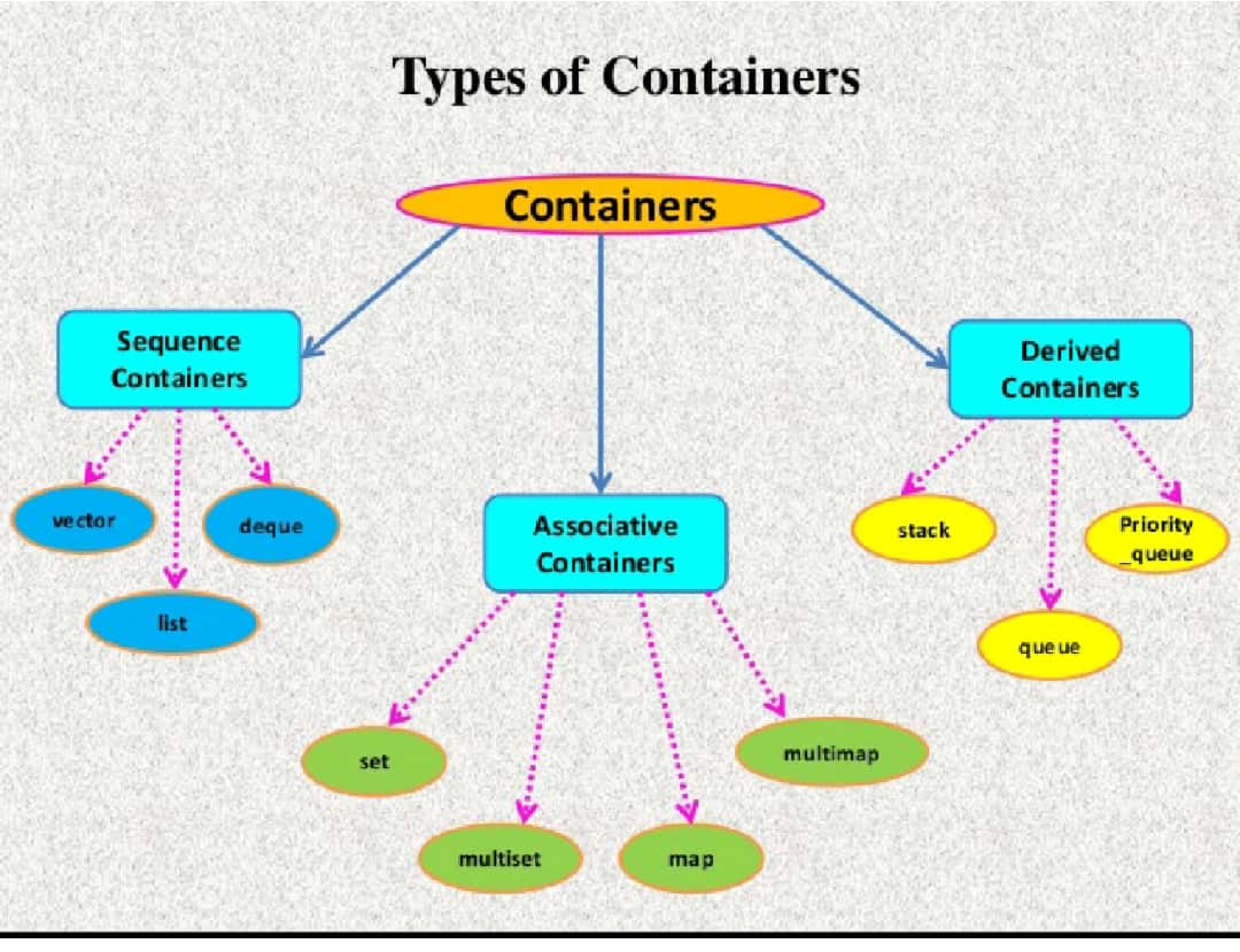
Container

Sequential

Associative

Container Adapters

Scanned by CamScanner



Types of Containers

Containers

Sequence Containers

vector

deque

list

Associative Containers

set

multiset

map

multimap

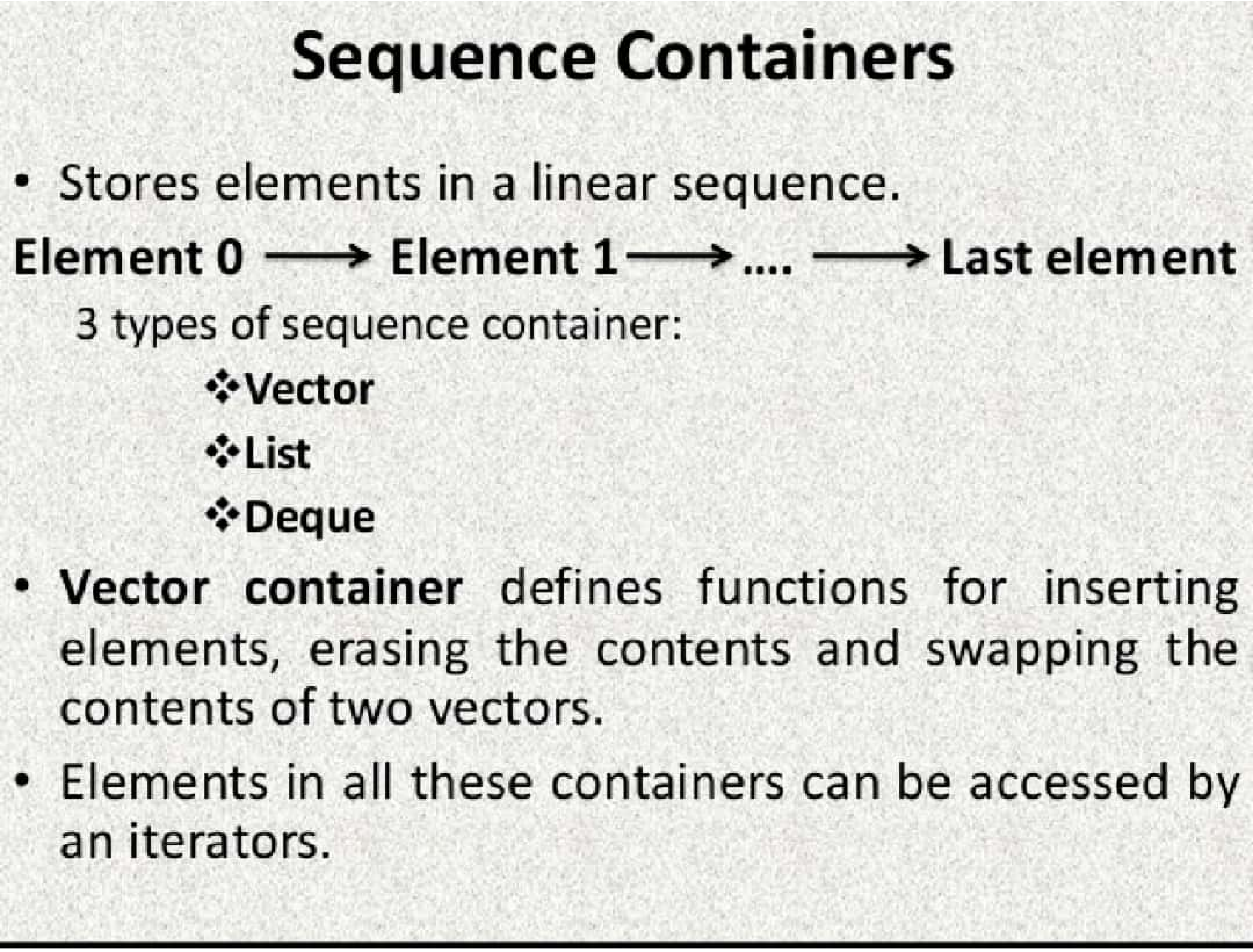
Derived Containers

stack

Priority _queue

queue

Scanned by CamScanner



Sequence Containers

- Stores elements in a linear sequence.

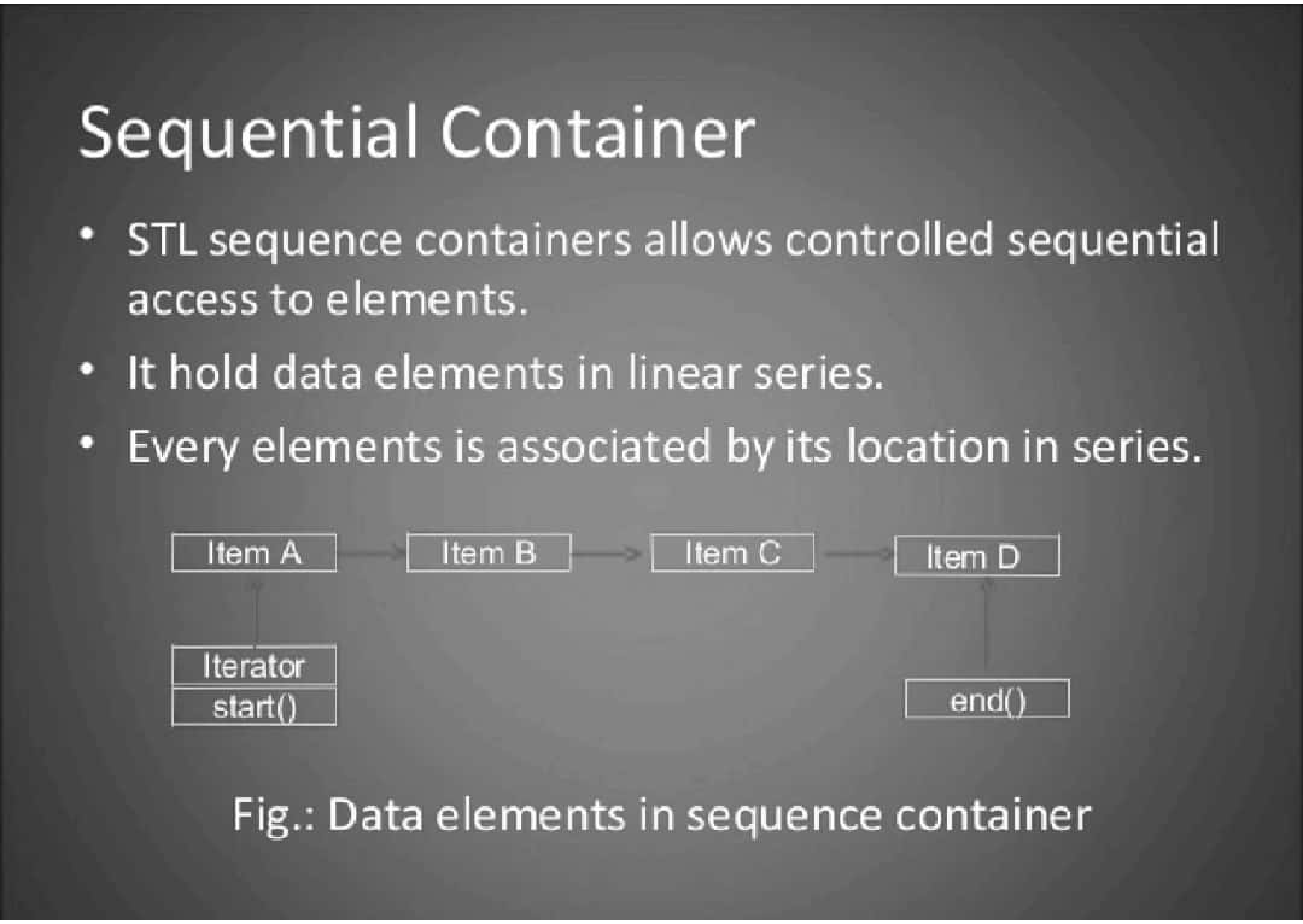
Element 0 → Element 1 → .... → Last element

3 types of sequence container:

- **Vector**
- **List**
- **Deque**

- **Vector container** defines functions for inserting elements, erasing the contents and swapping the contents of two vectors.
- Elements in all these containers can be accessed by an iterators.

Scanned by CamScanner

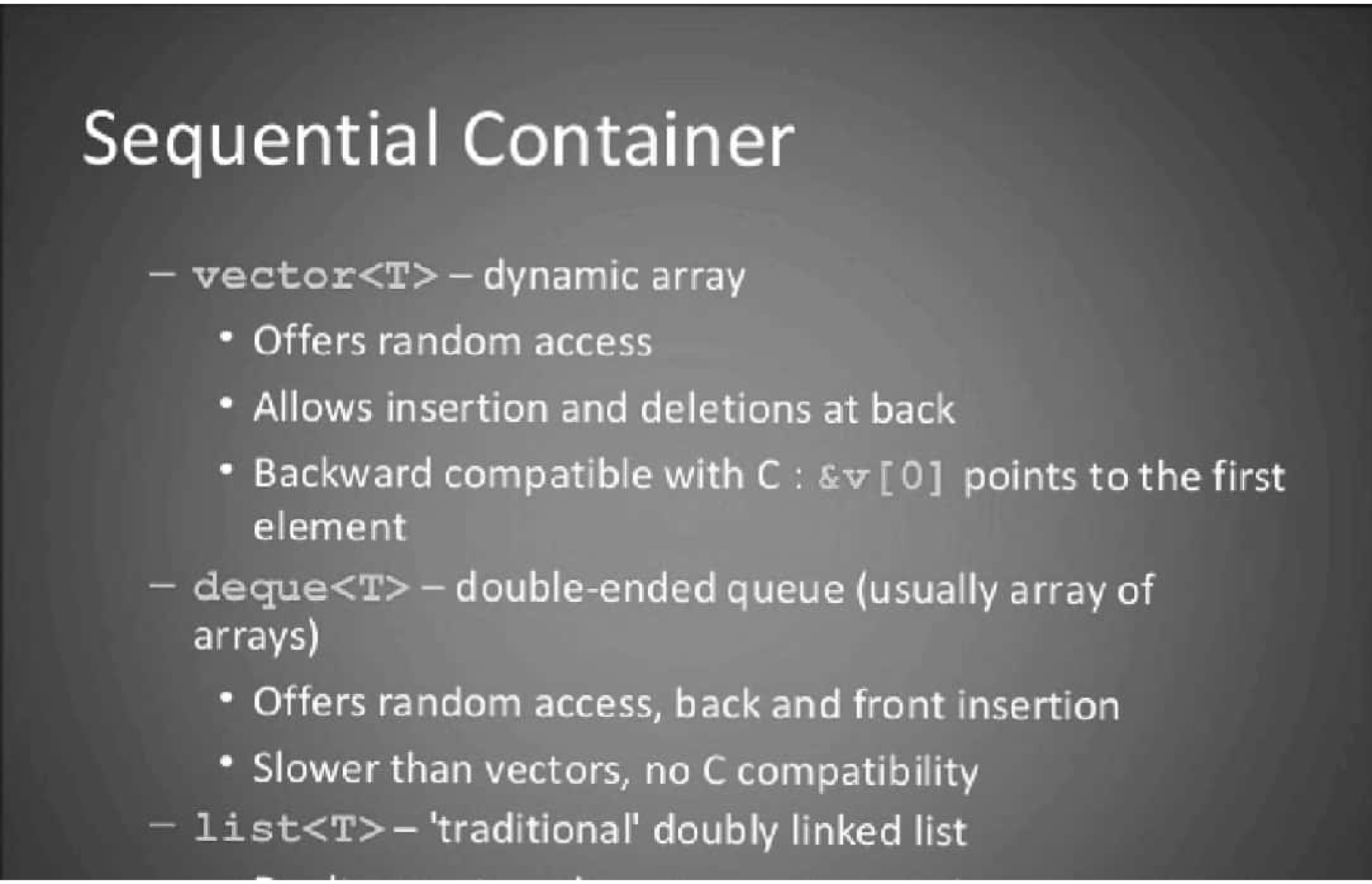


Sequential Container

- STL sequence containers allows controlled sequential access to elements.
- It hold data elements in linear series.
- Every elements is associated by its location in series.

Fig.: Data elements in sequence container

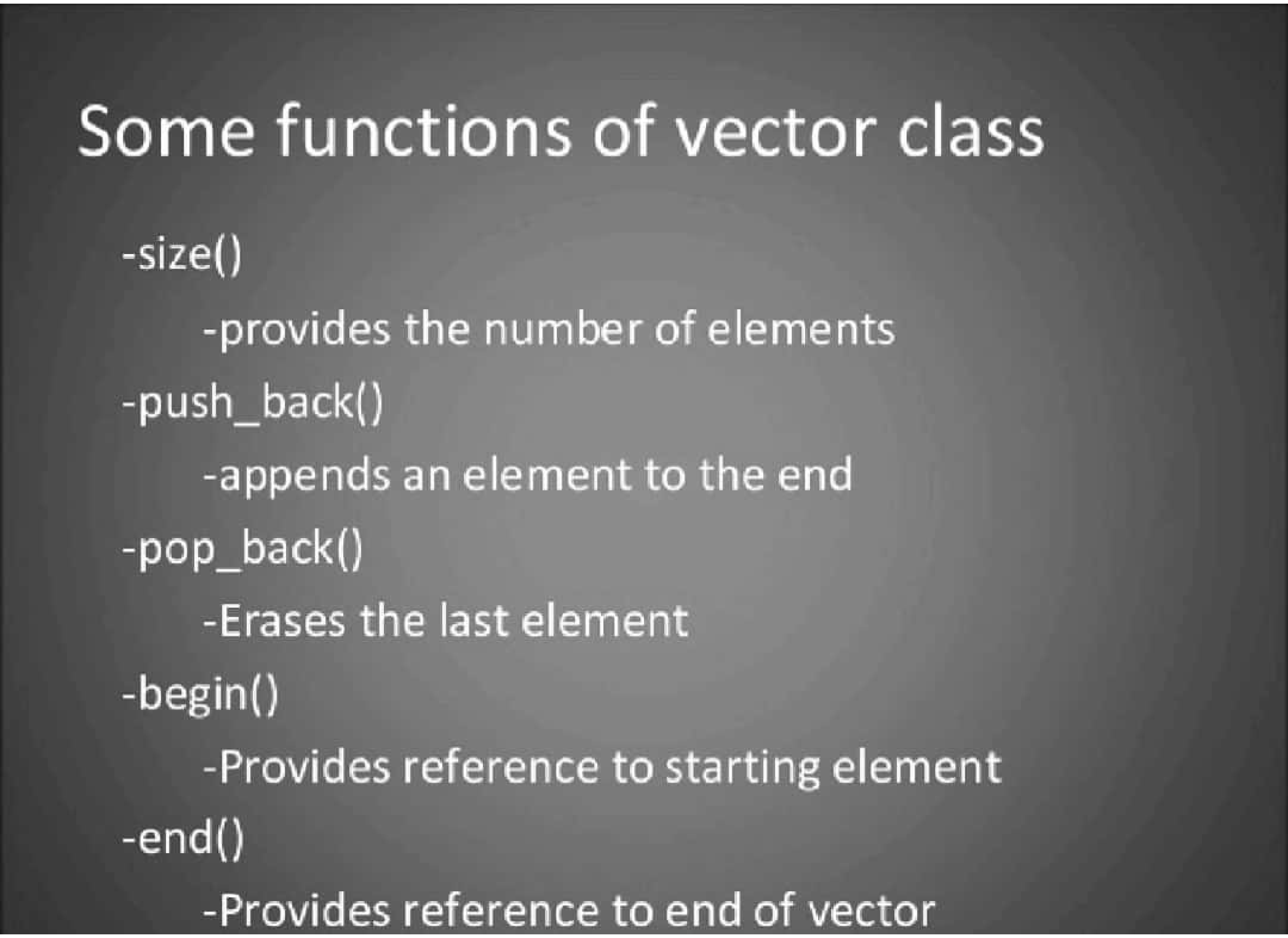
Scanned by CamScanner



Sequential Container

- `vector<T>` – dynamic array
  - Offers random access
  - Allows insertion and deletions at back
  - Backward compatible with C : `&v[0]` points to the first element
- `deque<T>` – double-ended queue (usually array of arrays)
  - Offers random access, back and front insertion
  - Slower than vectors, no C compatibility
- `list<T>` – 'traditional' doubly linked list

Scanned by CamScanner



Some functions of vector class

-size()

-provides the number of elements

-push_back()

-appends an element to the end

-pop_back()

-Erases the last element

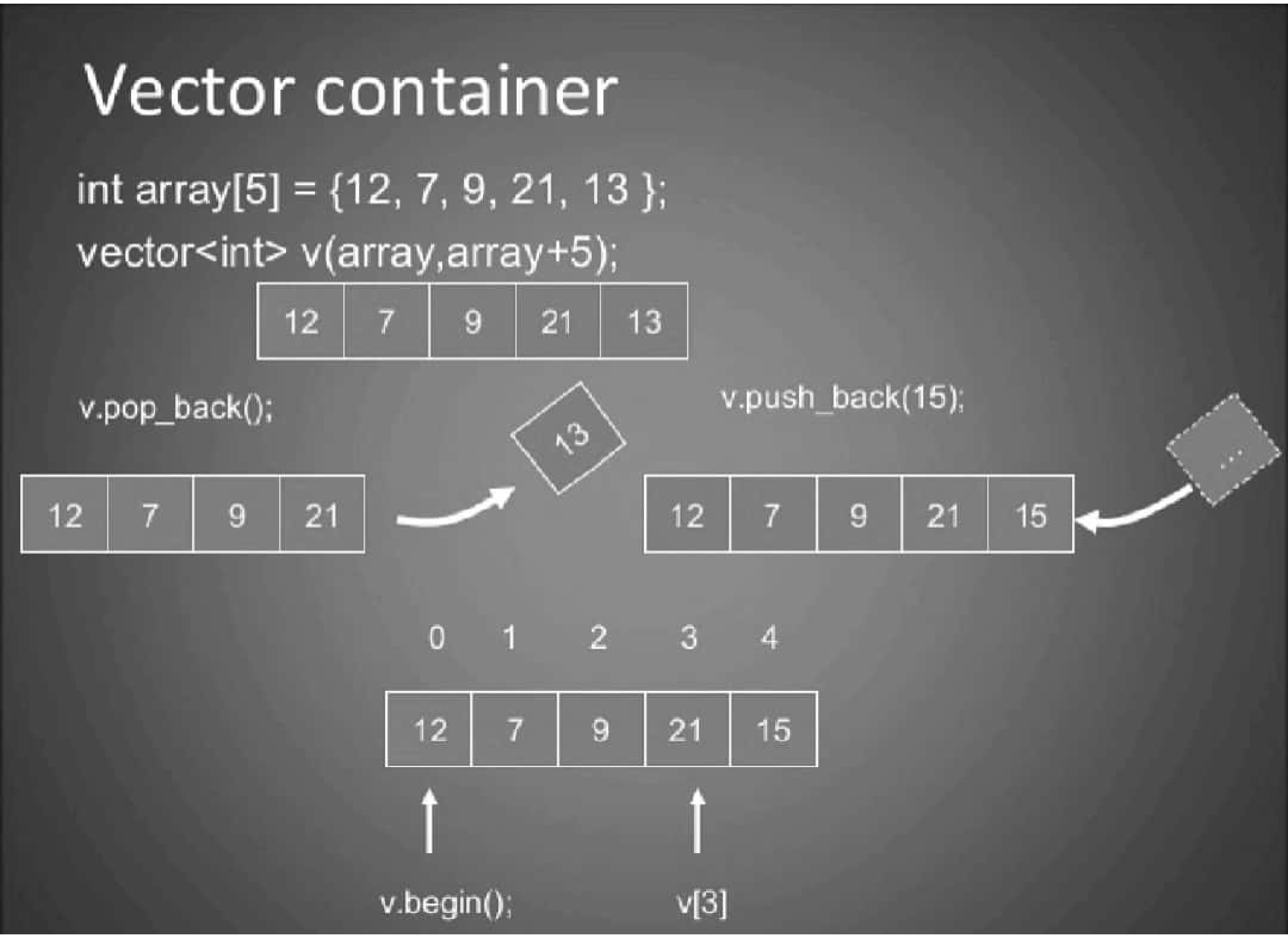
-begin()

-Provides reference to starting element

-end()

-Provides reference to end of vector

Scanned by CamScanner



Vector container

```
int array[5] = {12, 7, 9, 21, 13 };
vector<int> v(array,array+5);
```

12	7	9	21	13

v.pop_back();

12	7	9	21

v.push_back(15);

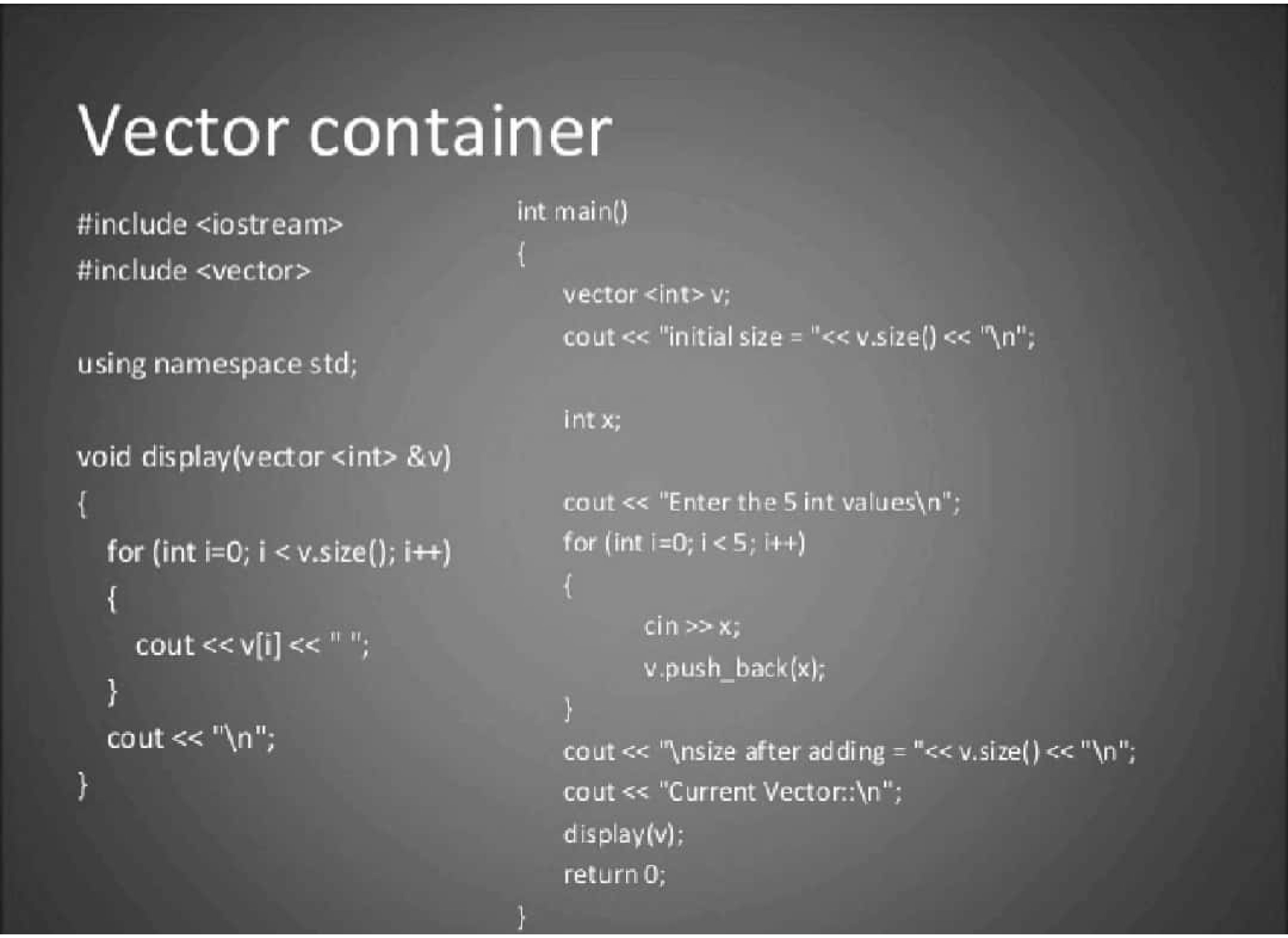
12	7	9	21	15

0	1	2	3	4
12	7	9	21	15

v.begin();

v[3]

Scanned by CamScanner

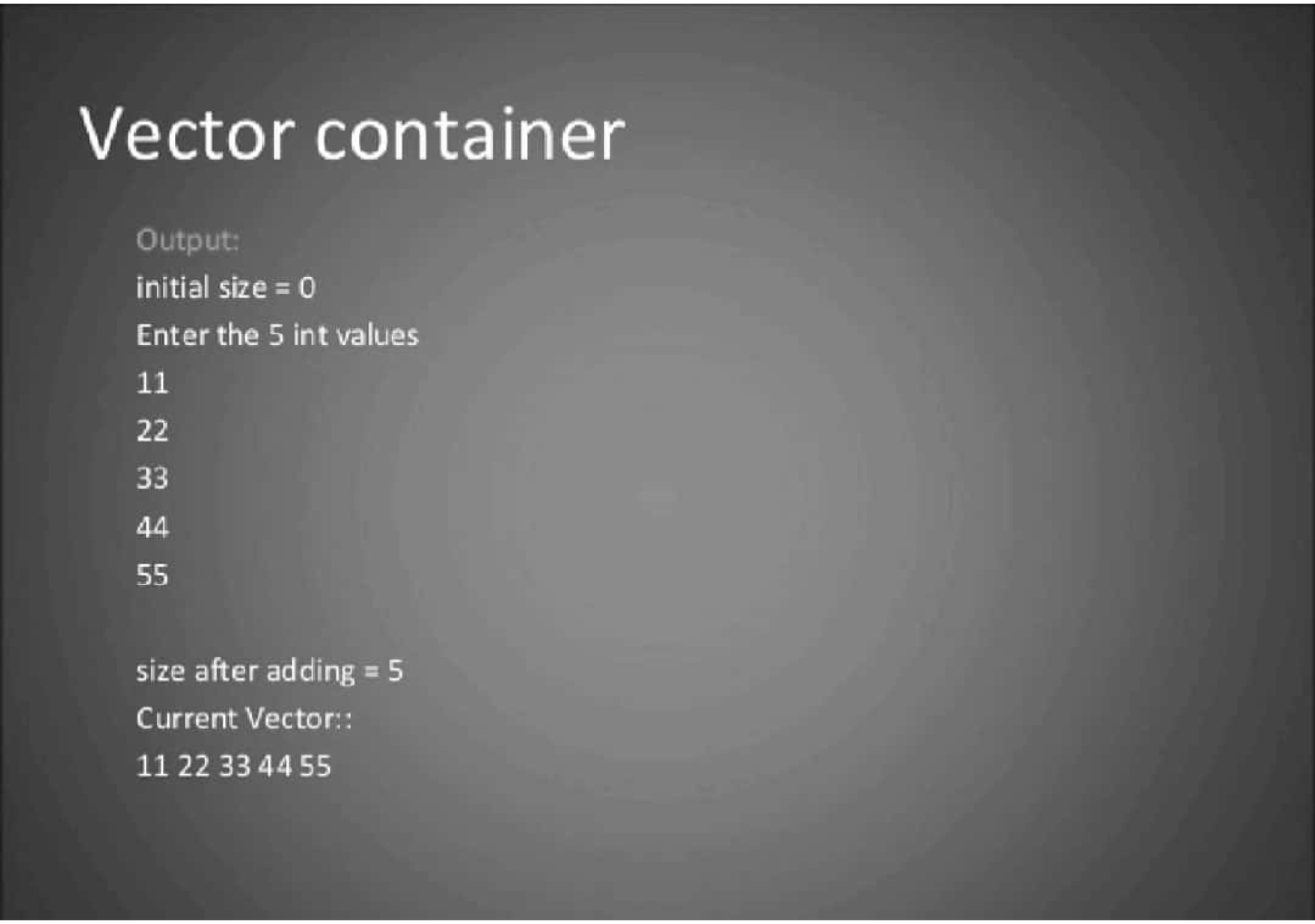


Vector container

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

using namespace std;

void display(vector <int> &v)
{
   for (int i=0; i < v.size(); i++)
   {
      cout << v[i] << " ";
   }
   cout << "\n";
}
```

```
int main()
{
      vector <int> v;
      cout << "initial size = "<< v.size() << "\n";

      int x;

      cout << "Enter the 5 int values\n";
      for (int i=0; i < 5; i++)
      {
            cin >> x;
            v.push_back(x);
      }
      cout << "\nsize after adding = "<< v.size() << "\n";
      cout << "Current Vector::\n";
      display(v);
      return 0;
}
```

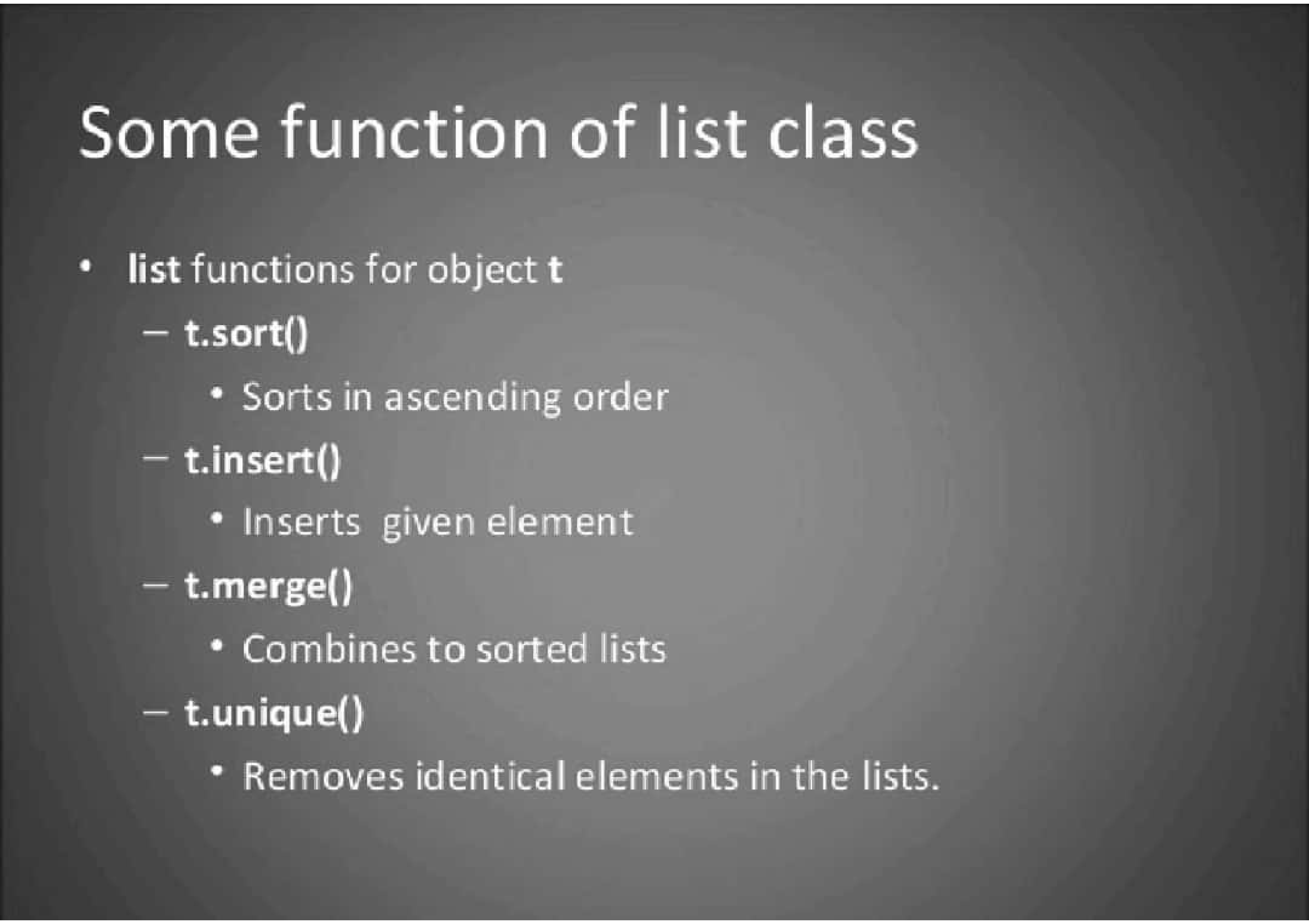
Scanned by CamScanner



Vector container

```
Output:
initial size = 0
Enter the 5 int values
11
22
33
44
55

size after adding = 5
Current Vector::
11 22 33 44 55
```

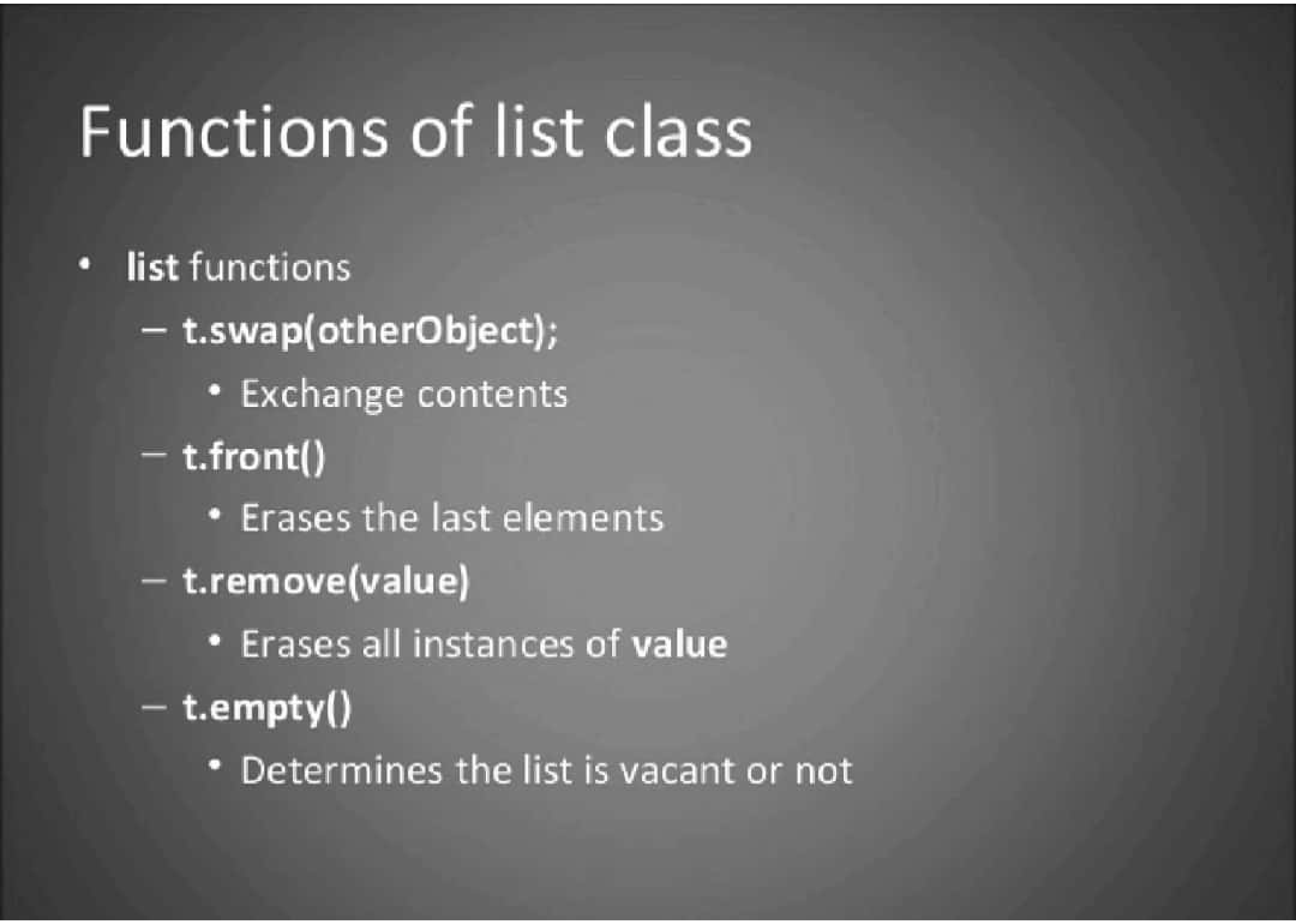
Scanned by CamScanner



Some function of list class

- **list** functions for object **t**
  - **t.sort()**
    - Sorts in ascending order
  - **t.insert()**
    - Inserts given element
  - **t.merge()**
    - Combines to sorted lists
  - **t.unique()**
    - Removes identical elements in the lists.

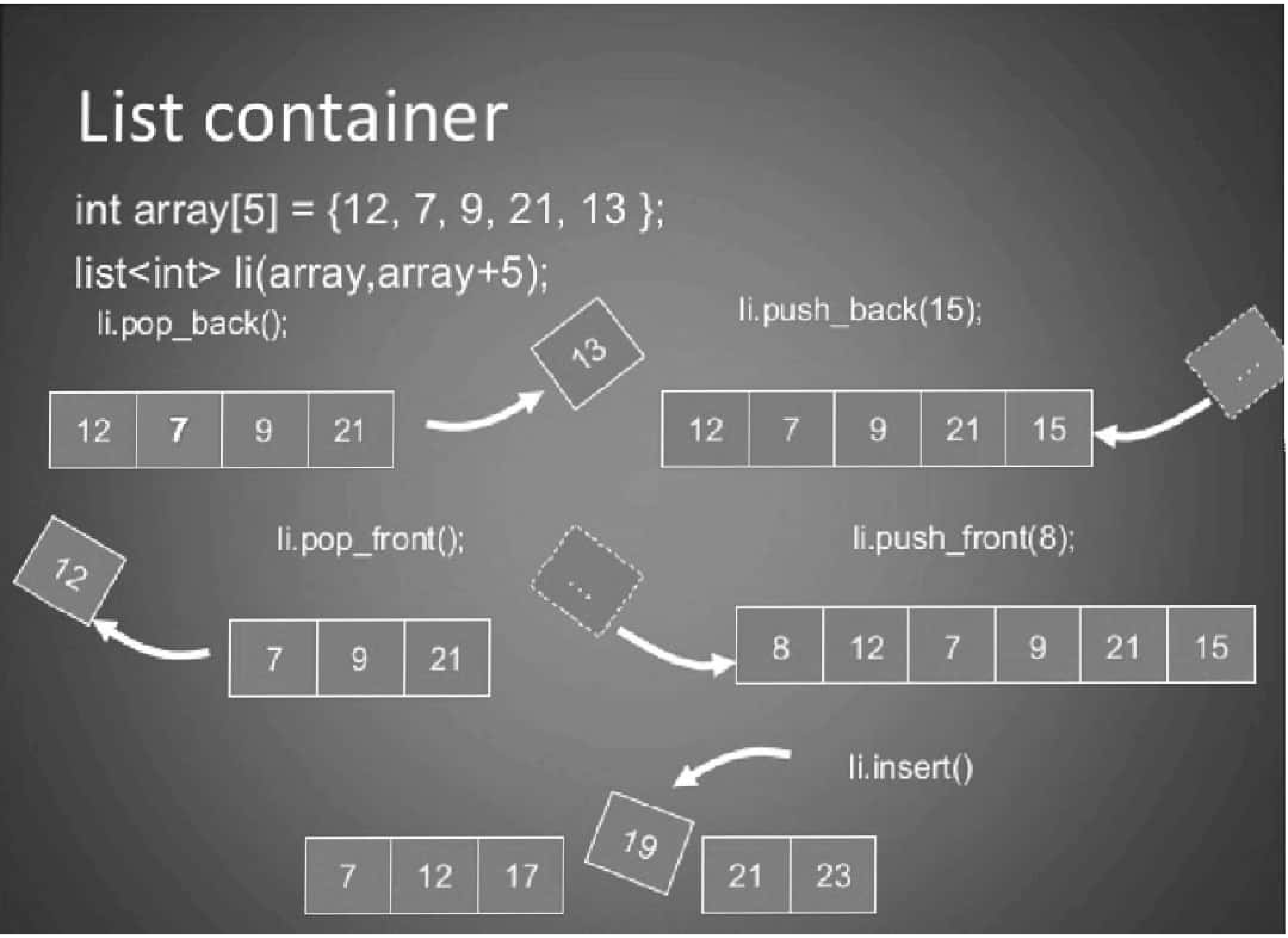
Scanned by CamScanner



Functions of list class

- **list** functions
  - **t.swap(otherObject);**
    - Exchange contents
  - **t.front()**
    - Erases the last elements
  - **t.remove(value)**
    - Erases all instances of **value**
  - **t.empty()**
    - Determines the list is vacant or not

Scanned by CamScanner



List container

```
int array[5] = {12, 7, 9, 21, 13 };
list<int> li(array,array+5);
```

li.pop_back();

12	7	9	21

li.push_back(15);

12	7	9	21	15

li.pop_front();

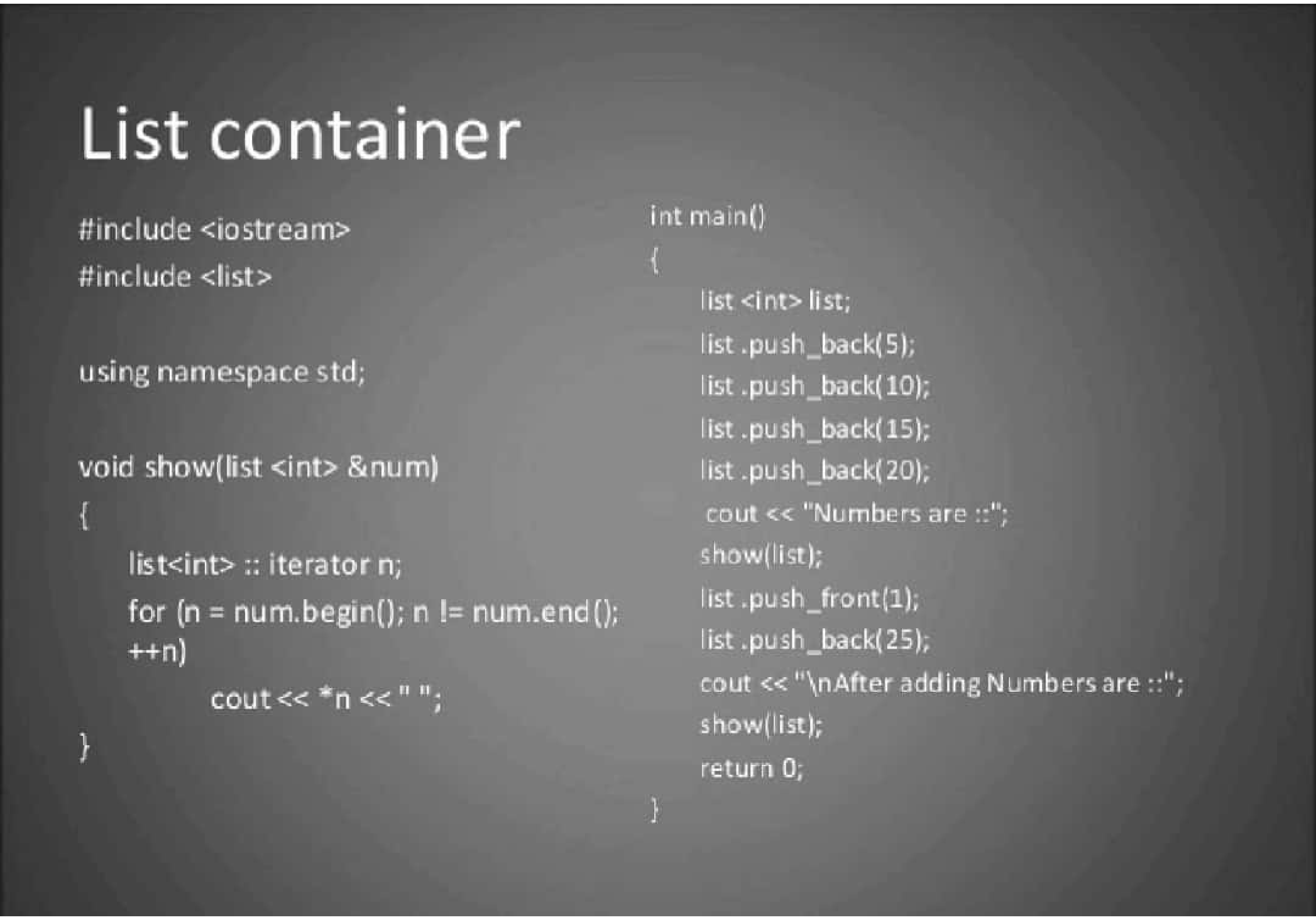
7	9	21

li.push_front(8);

8	12	7	9	21	15

li.insert()

7	12	17		21	23

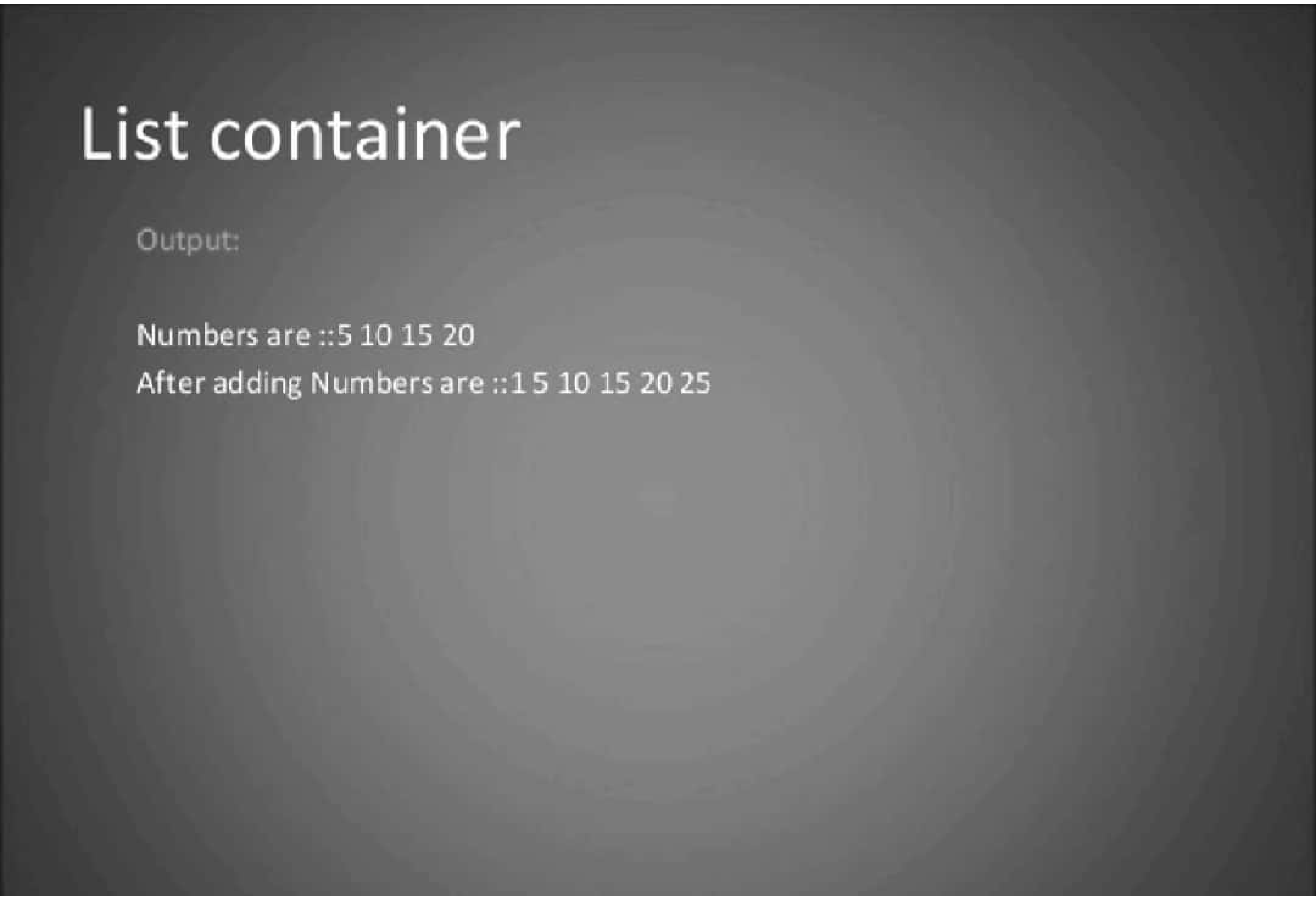


List container

```
#include <iostream>
#include <list>

using namespace std;

void show(list <int> &num)
{
    list<int> :: iterator n;
    for (n = num.begin(); n != num.end();
    ++n)
        cout << *n << " ";
}
```

```
int main()
{
    list <int> list;
    list .push_back(5);
    list .push_back(10);
    list .push_back(15);
    list .push_back(20);
     cout << "Numbers are ::";
    show(list);
    list .push_front(1);
    list .push_back(25);
    cout << "\nAfter adding Numbers are ::";
    show(list);
    return 0;
}
```

Scanned by CamScanner

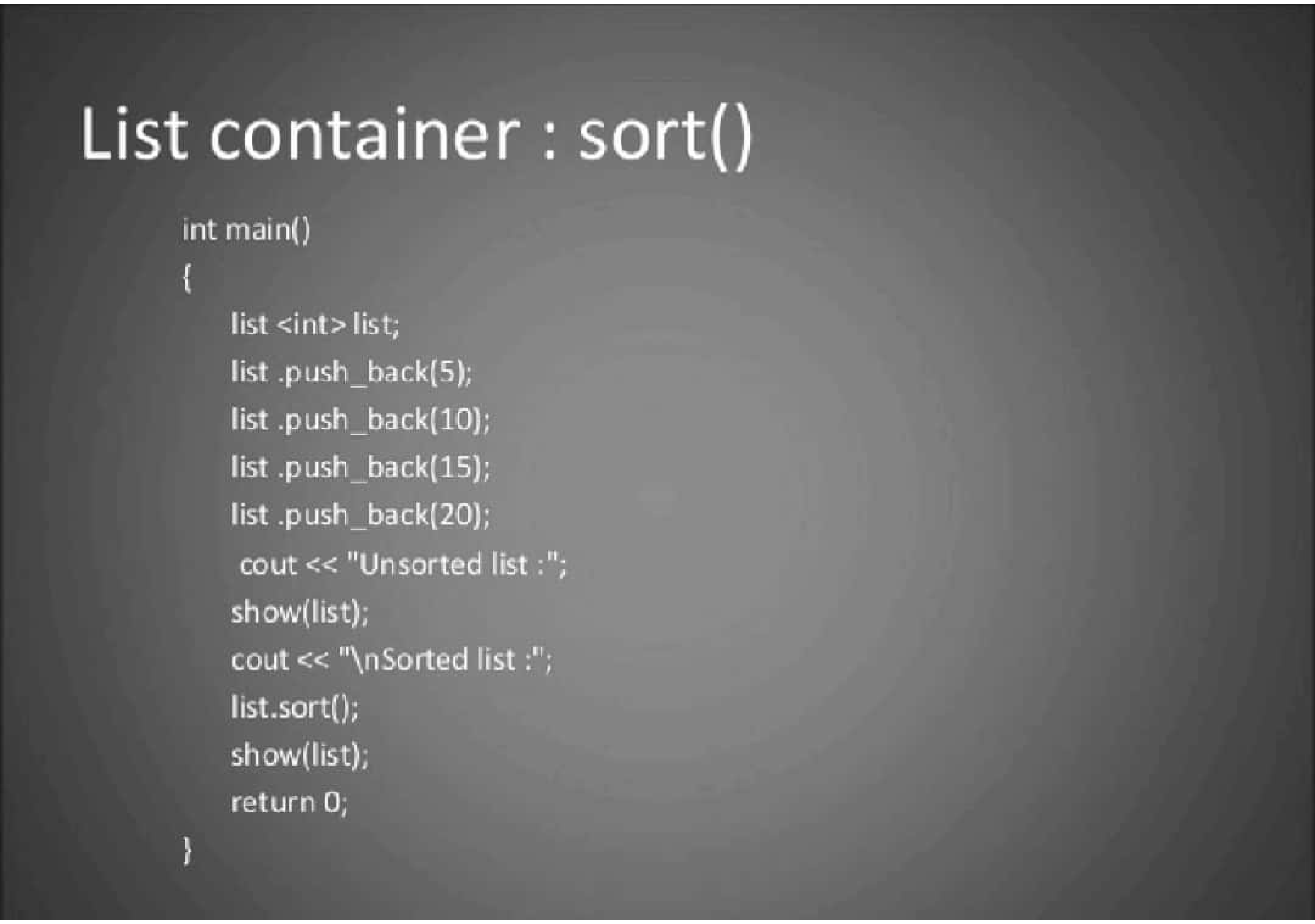


List container

Output:

```
Numbers are ::5 10 15 20
After adding Numbers are ::1 5 10 15 20 25
```

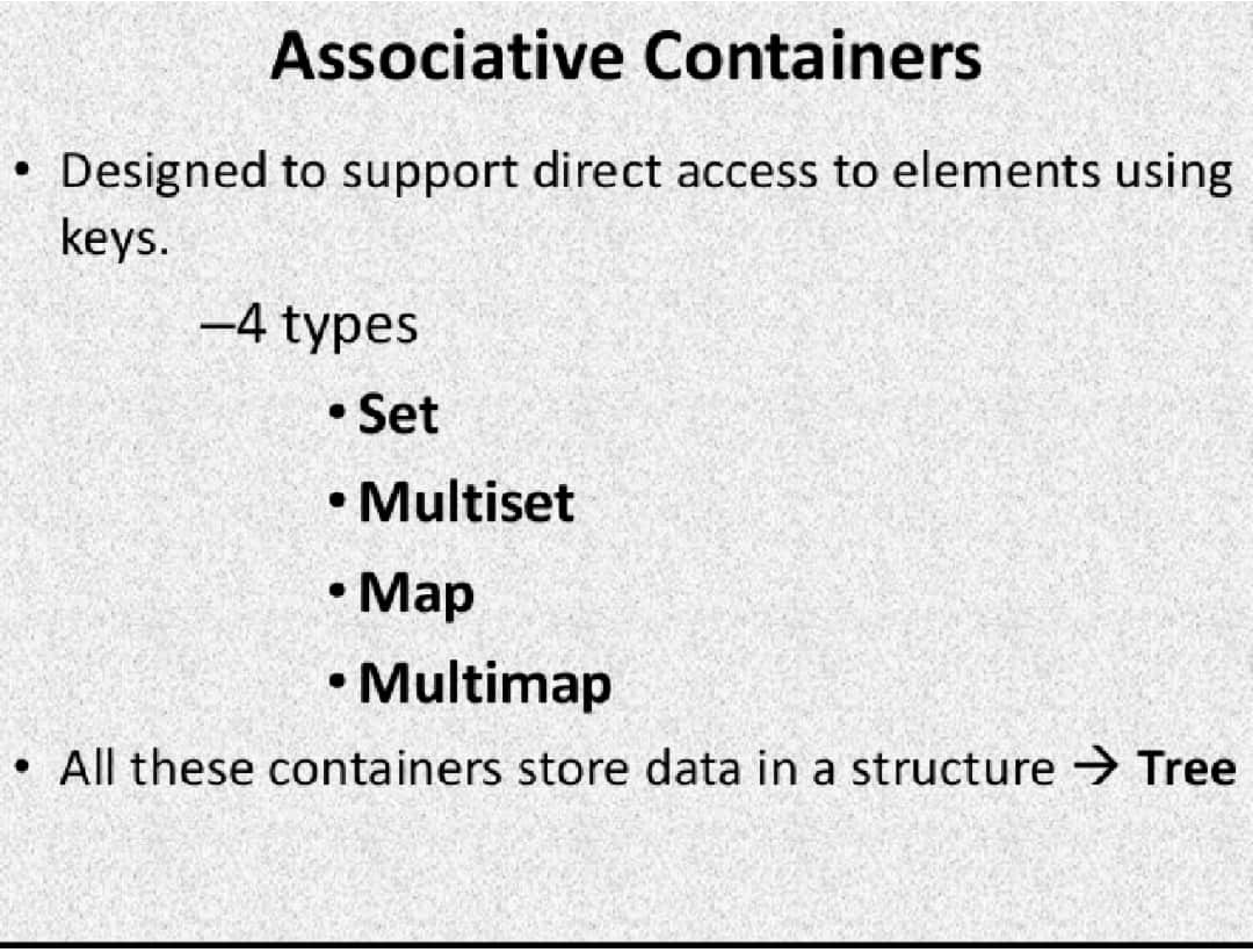
Scanned by CamScanner



List container : sort()

```
int main()
{
    list <int> list;
    list .push_back(5);
    list .push_back(10);
    list .push_back(15);
    list .push_back(20);
     cout << "Unsorted list :";
    show(list);
    cout << "\nSorted list :";
    list.sort();
    show(list);
    return 0;
}
```

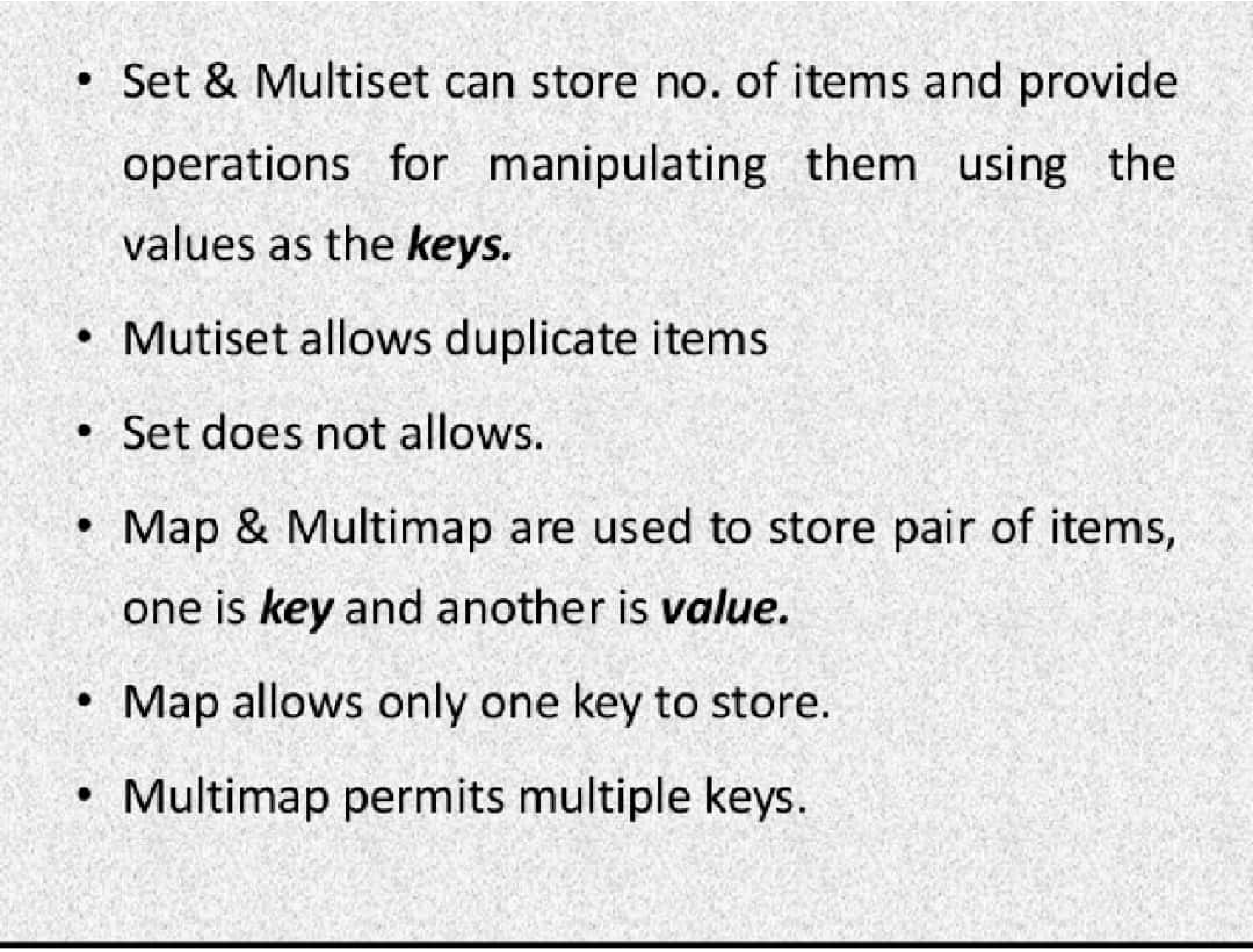
Scanned by CamScanner



Associative Containers

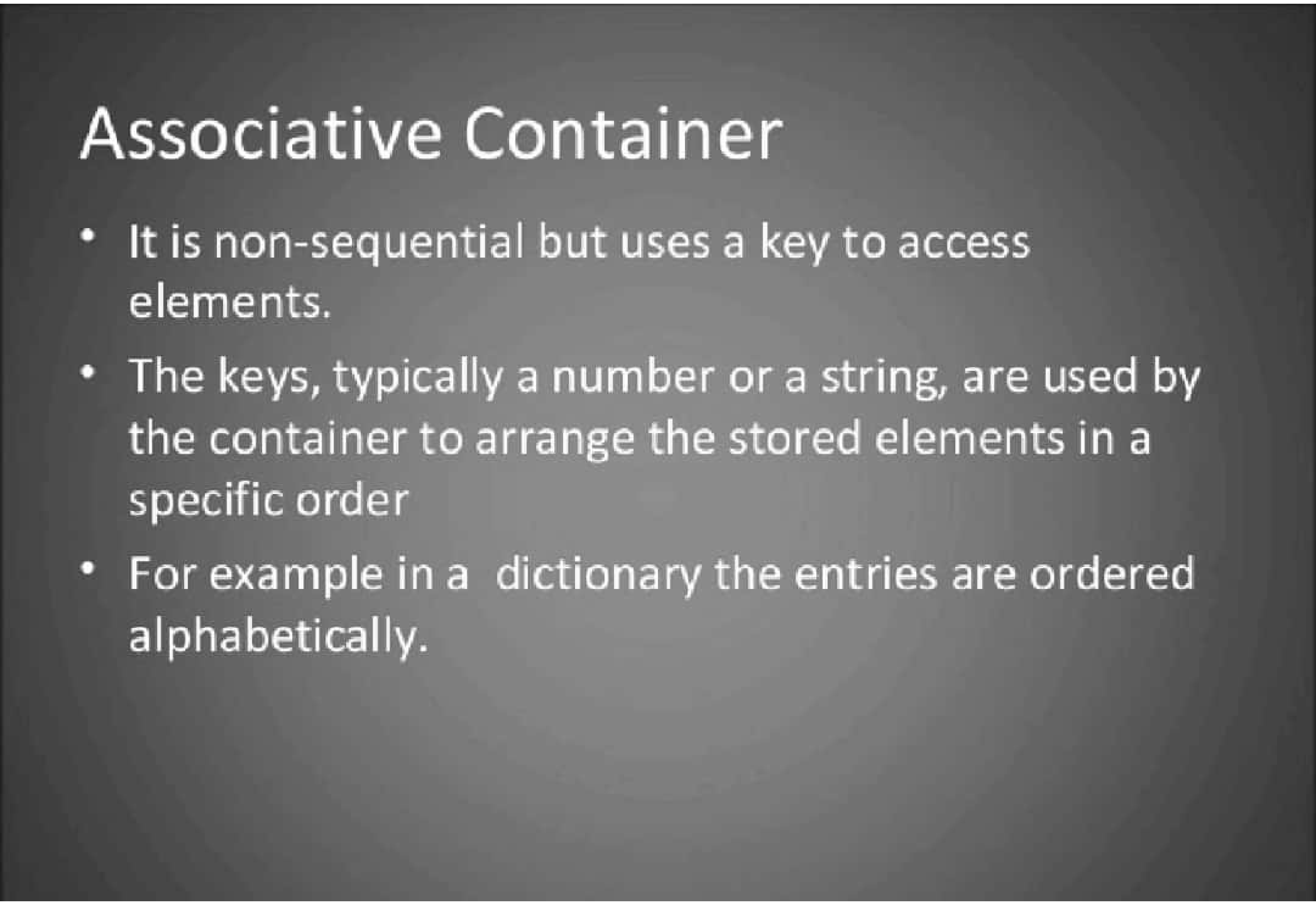
- Designed to support direct access to elements using keys.
  - –4 types
    - **Set**
    - **Multiset**
    - **Map**
    - **Multimap**
- All these containers store data in a structure → **Tree**

Scanned by CamScanner



- Set & Multiset can store no. of items and provide operations for manipulating them using the values as the ***keys.***
- Mutiset allows duplicate items
- Set does not allows.
- Map & Multimap are used to store pair of items, one is ***key*** and another is ***value.***
- Map allows only one key to store.
- Multimap permits multiple keys.

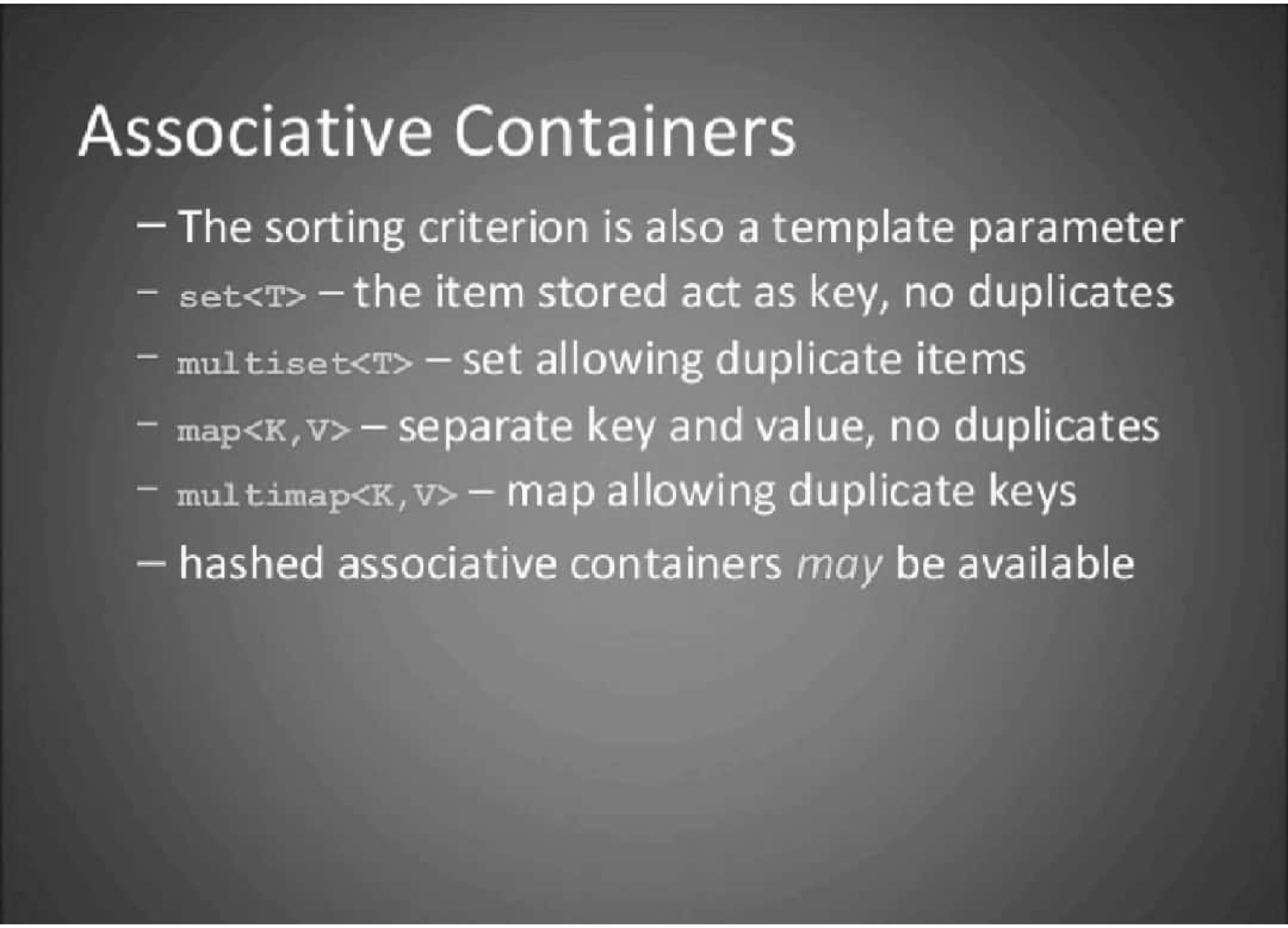
Scanned by CamScanner



Associative Container

- It is non-sequential but uses a key to access elements.
- The keys, typically a number or a string, are used by the container to arrange the stored elements in a specific order
- For example in a dictionary the entries are ordered alphabetically.

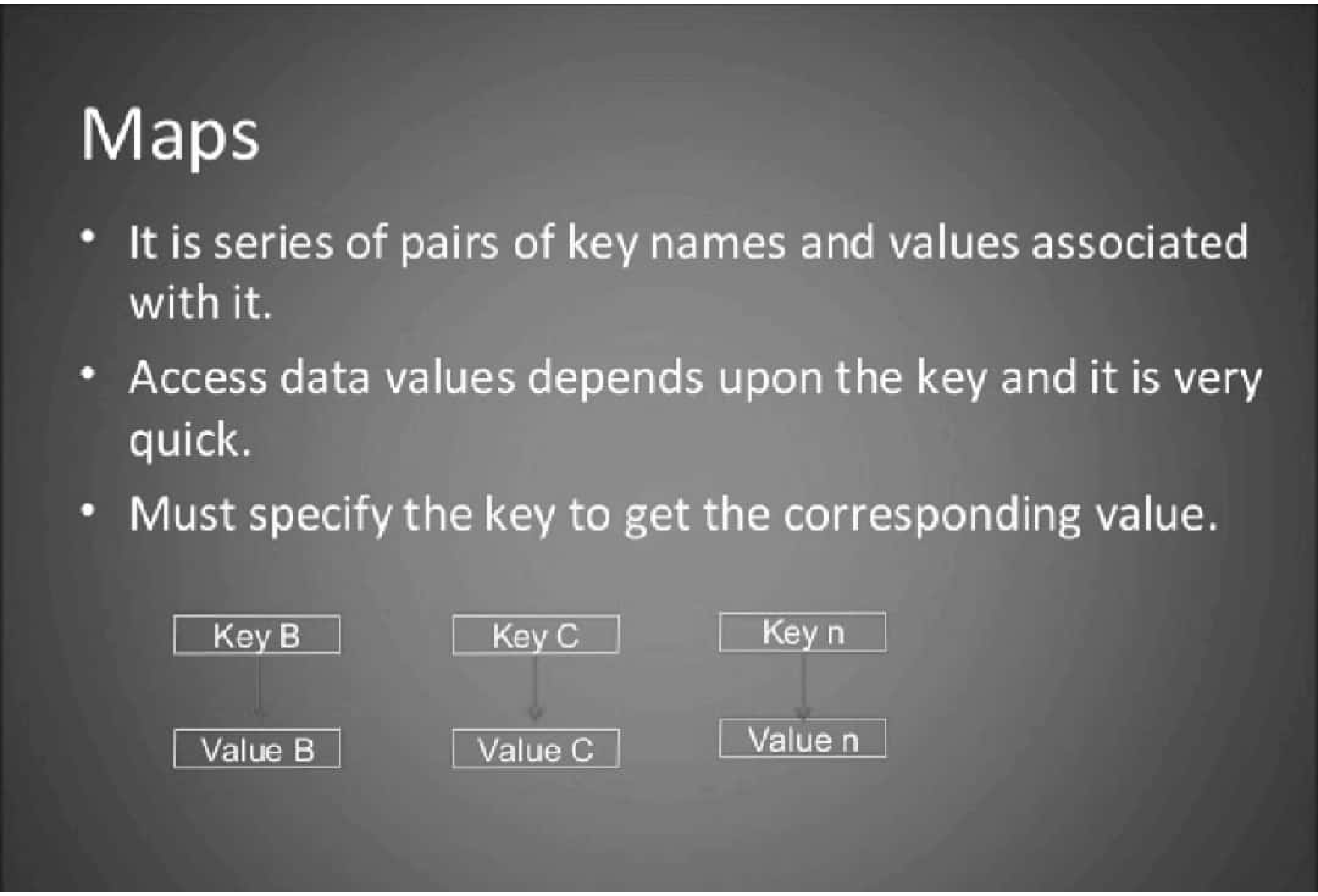
Scanned by CamScanner



Associative Containers

- The sorting criterion is also a template parameter
- `set<T>` – the item stored act as key, no duplicates
- `multiset<T>` – set allowing duplicate items
- `map<K,V>` – separate key and value, no duplicates
- `multimap<K,V>` – map allowing duplicate keys
- hashed associative containers *may* be available

Scanned by CamScanner

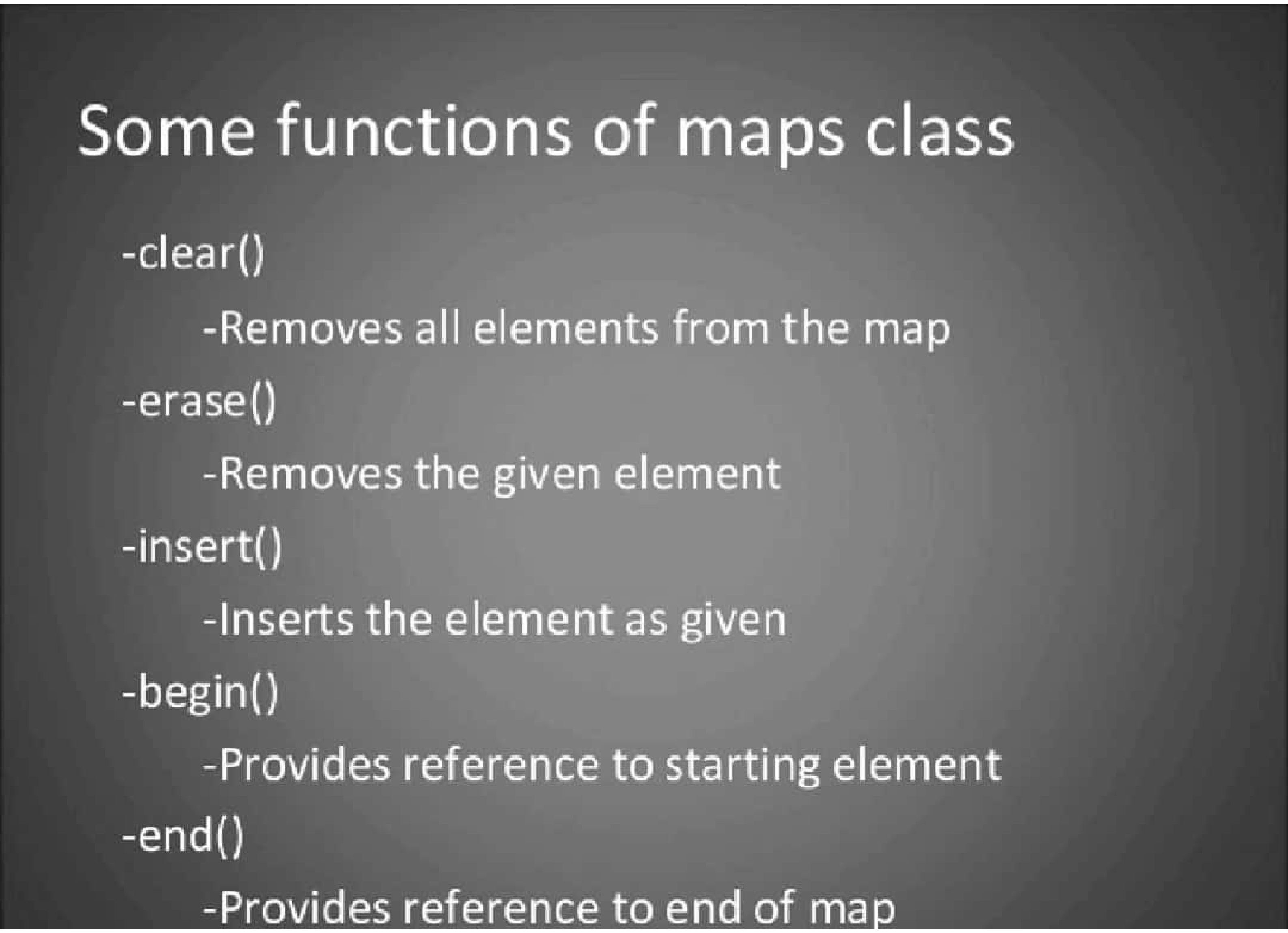


Maps

- It is series of pairs of key names and values associated with it.
- Access data values depends upon the key and it is very quick.
- Must specify the key to get the corresponding value.

Key B
Value B
Key C
Value C
Key n
Value n

Scanned by CamScanner



Some functions of maps class

-clear()

- -Removes all elements from the map

-erase()

- -Removes the given element

-insert()

- -Inserts the element as given

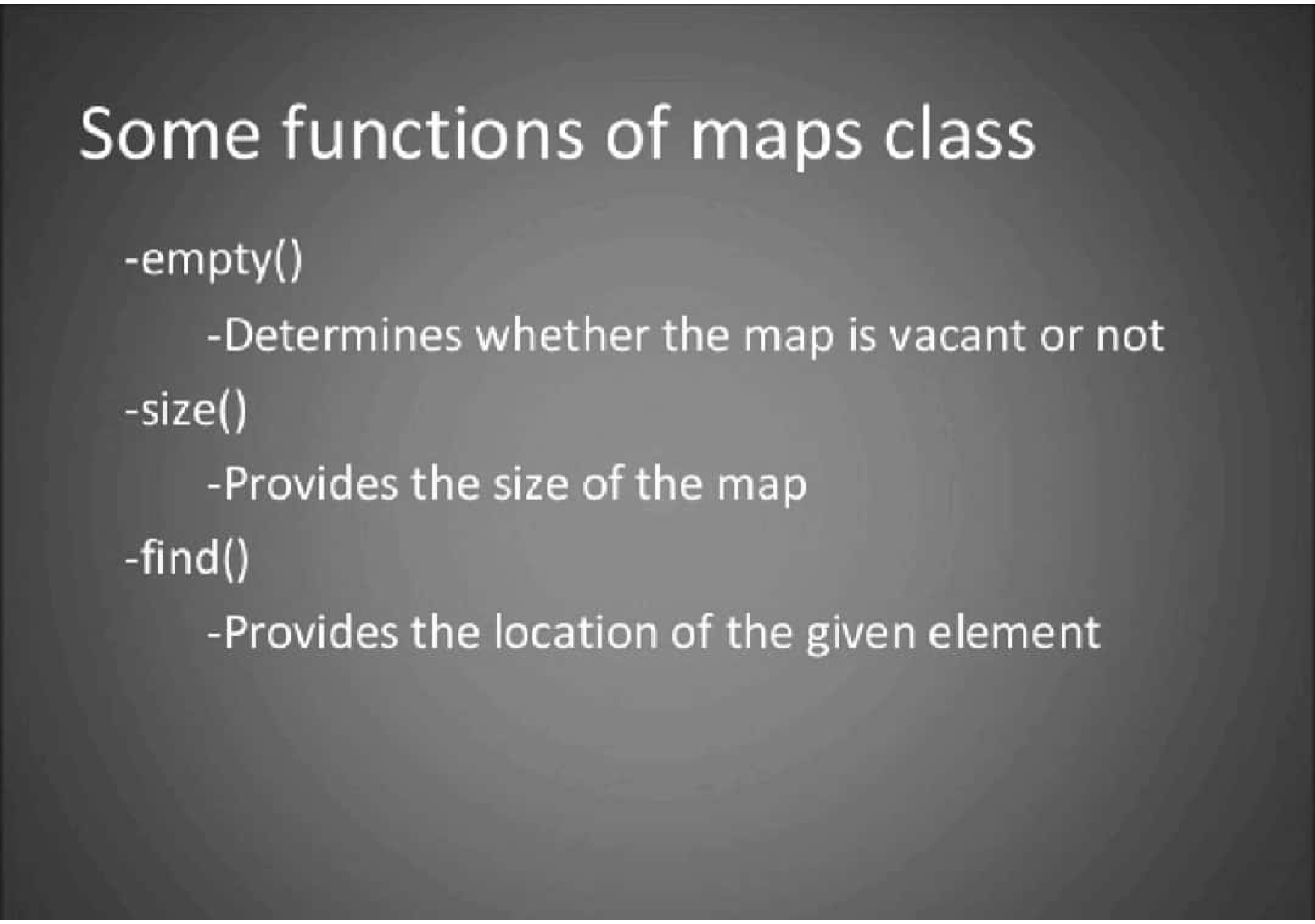
-begin()

- -Provides reference to starting element

-end()

- -Provides reference to end of map

Scanned by CamScanner



Some functions of maps class

-empty()

  -Determines whether the map is vacant or not

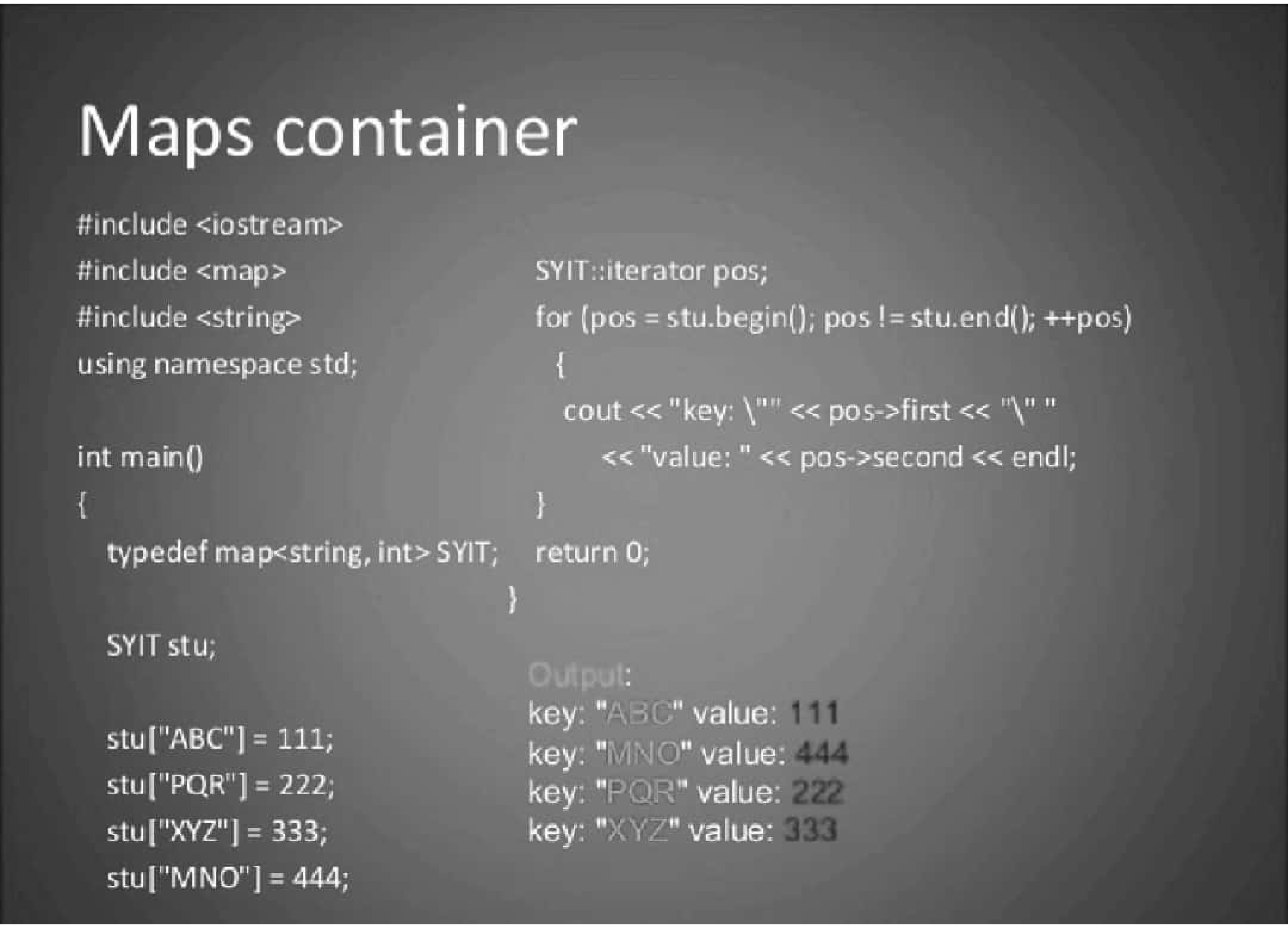
-size()

  -Provides the size of the map

-find()

  -Provides the location of the given element

Scanned by CamScanner



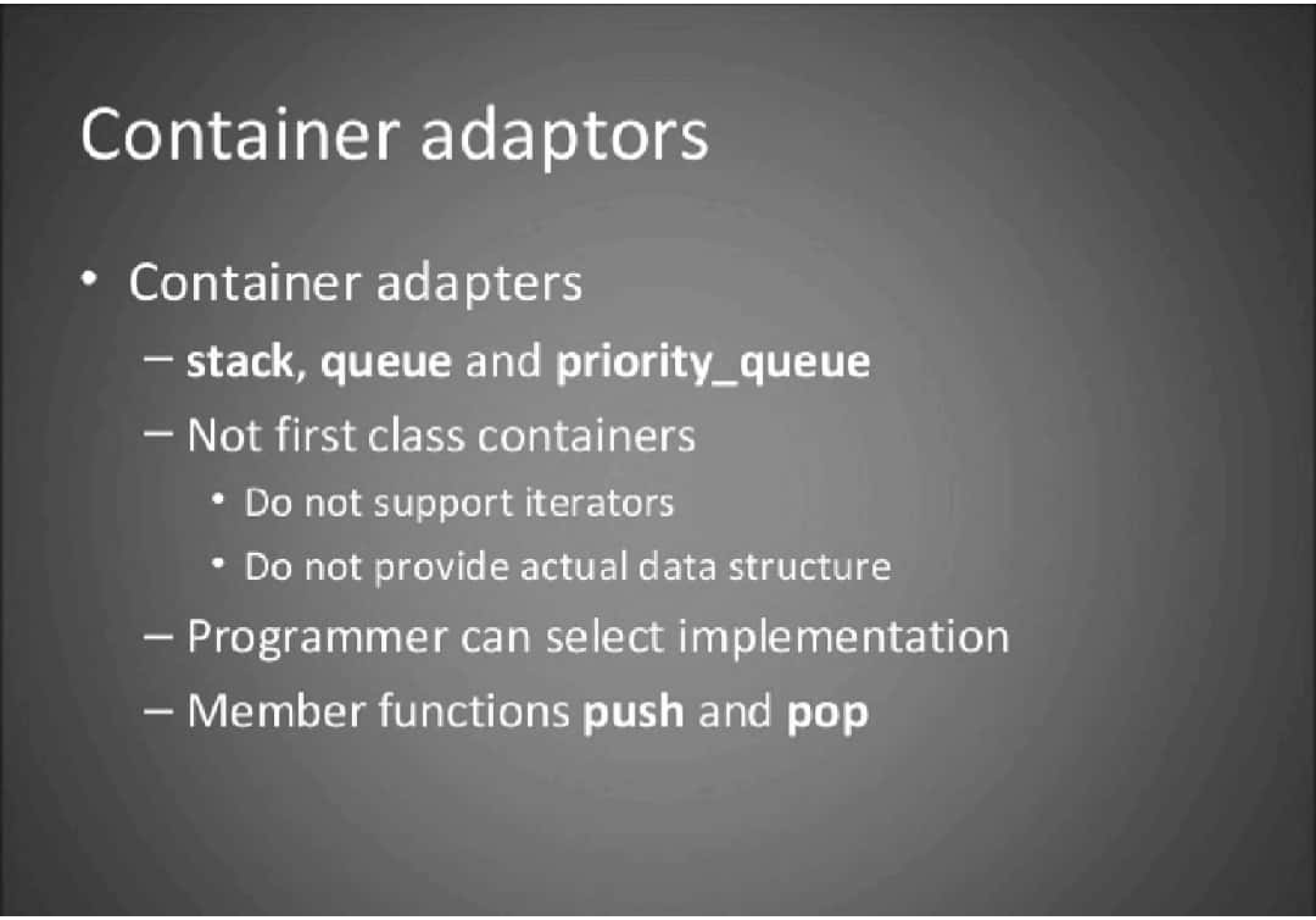
Maps container

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

int main()
{
   typedef map<string, int> SYIT;

   SYIT stu;

   stu["ABC"] = 111;
   stu["PQR"] = 222;
   stu["XYZ"] = 333;
   stu["MNO"] = 444;

   SYIT::iterator pos;
   for (pos = stu.begin(); pos != stu.end(); ++pos)
     {
      cout << "key: \"" << pos->first << "\" "
           << "value: " << pos->second << endl;
     }
   return 0;
}
```

Output:

```
key: "ABC" value: 111
key: "MNO" value: 444
key: "PQR" value: 222
key: "XYZ" value: 333
```

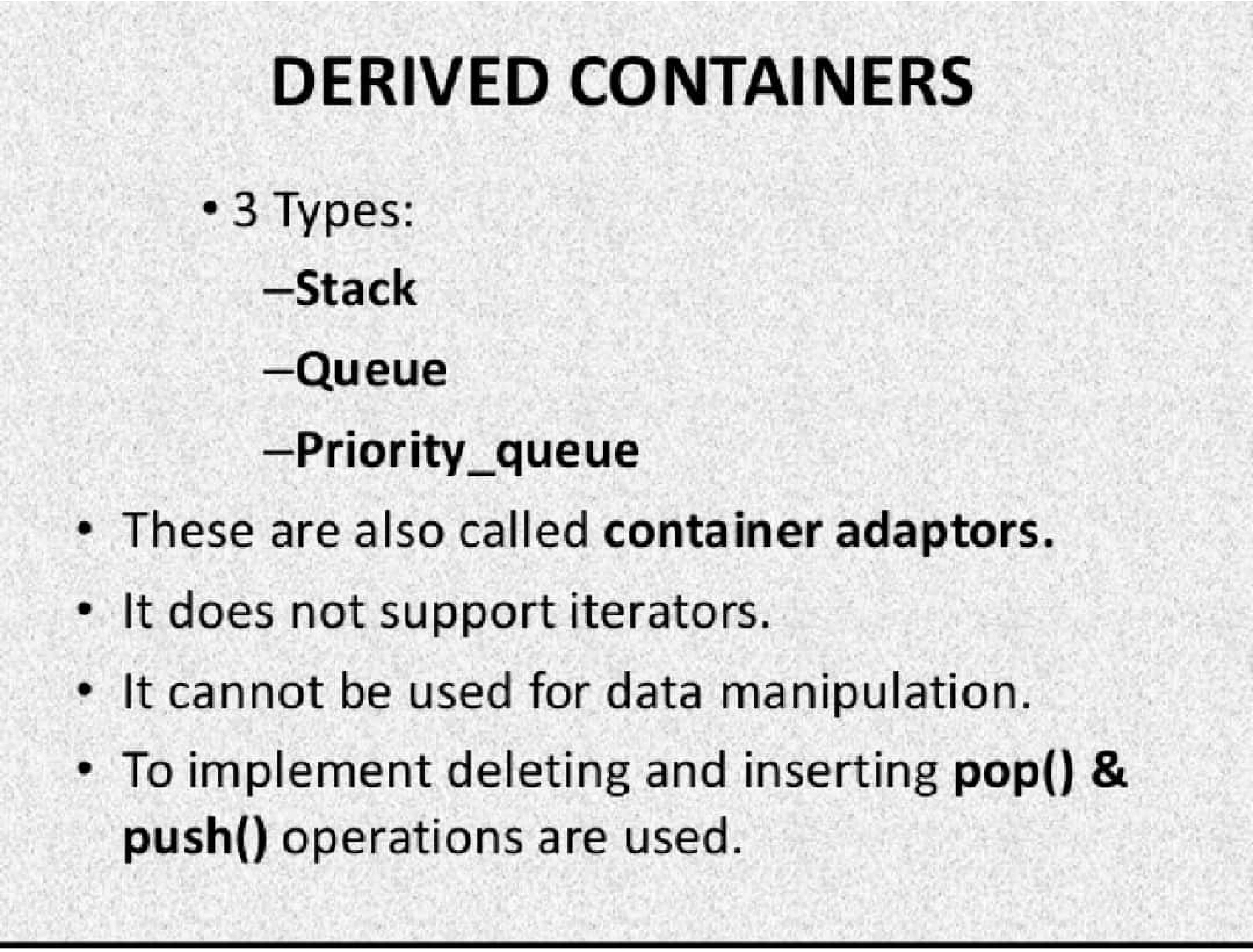
Scanned by CamScanner



Container adaptors

- Container adapters
  - **stack, queue** and **priority_queue**
  - Not first class containers
    - Do not support iterators
    - Do not provide actual data structure
  - Programmer can select implementation
  - Member functions **push** and **pop**

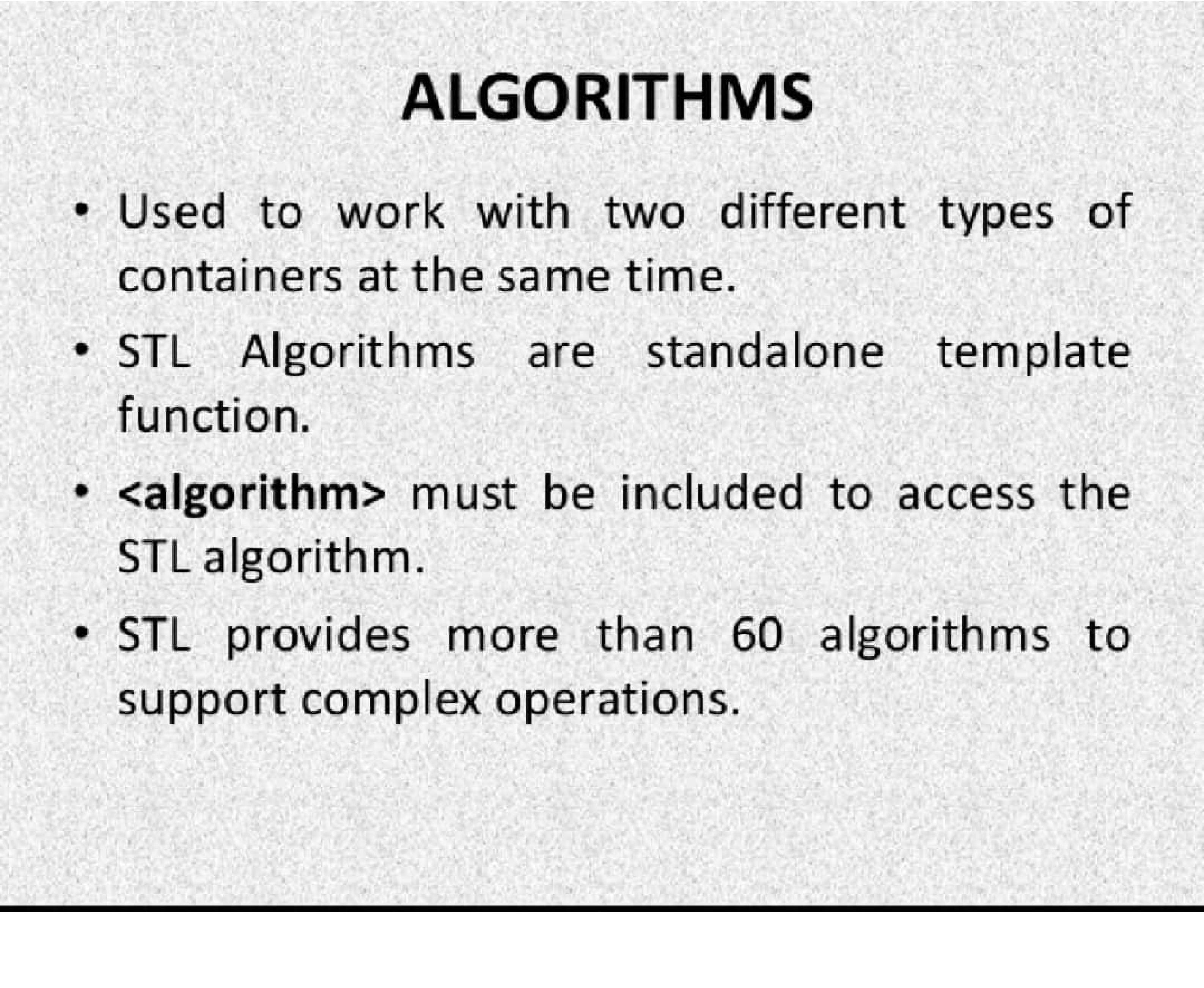
Scanned by CamScanner



DERIVED CONTAINERS

- 3 Types:
  - –**Stack**
  - –**Queue**
  - –**Priority_queue**
- These are also called **container adaptors.**
- It does not support iterators.
- It cannot be used for data manipulation.
- To implement deleting and inserting **pop() & push()** operations are used.

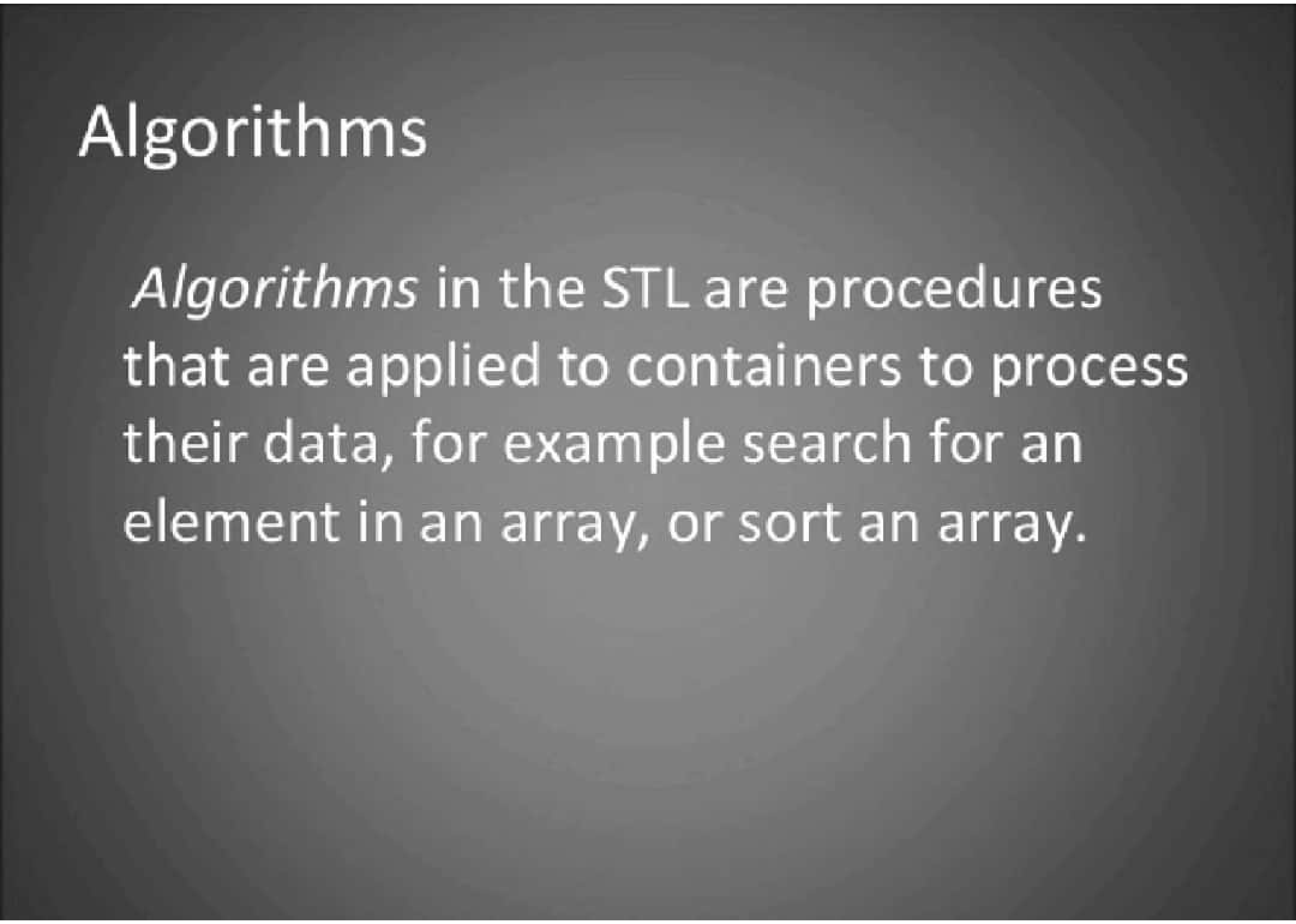
Scanned by CamScanner



ALGORITHMS

- Used to work with two different types of containers at the same time.
- STL Algorithms are standalone template function.
- **<algorithm>** must be included to access the STL algorithm.
- STL provides more than 60 algorithms to support complex operations.

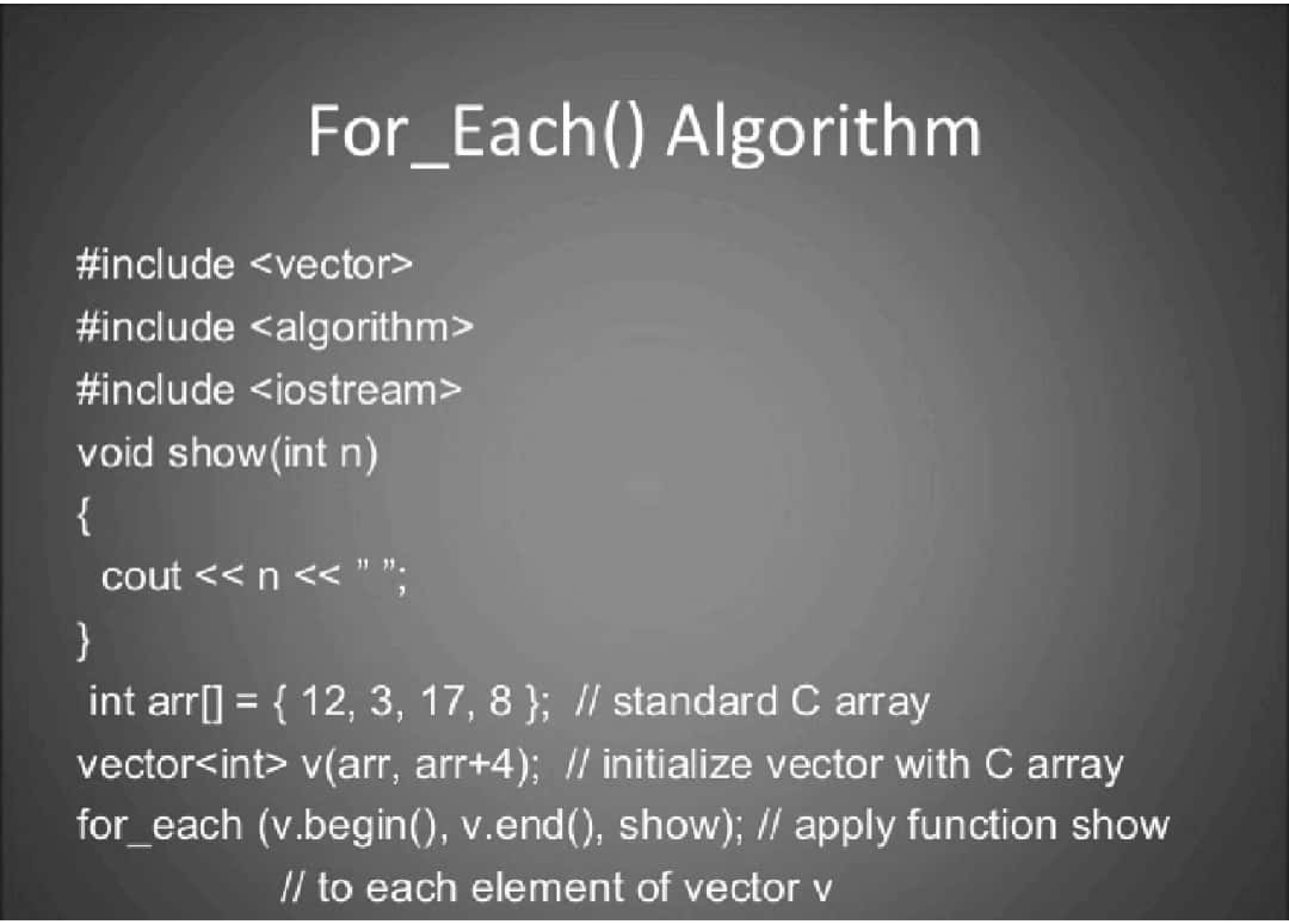
Scanned by CamScanner



Algorithms

Algorithms in the STL are procedures that are applied to containers to process their data, for example search for an element in an array, or sort an array.

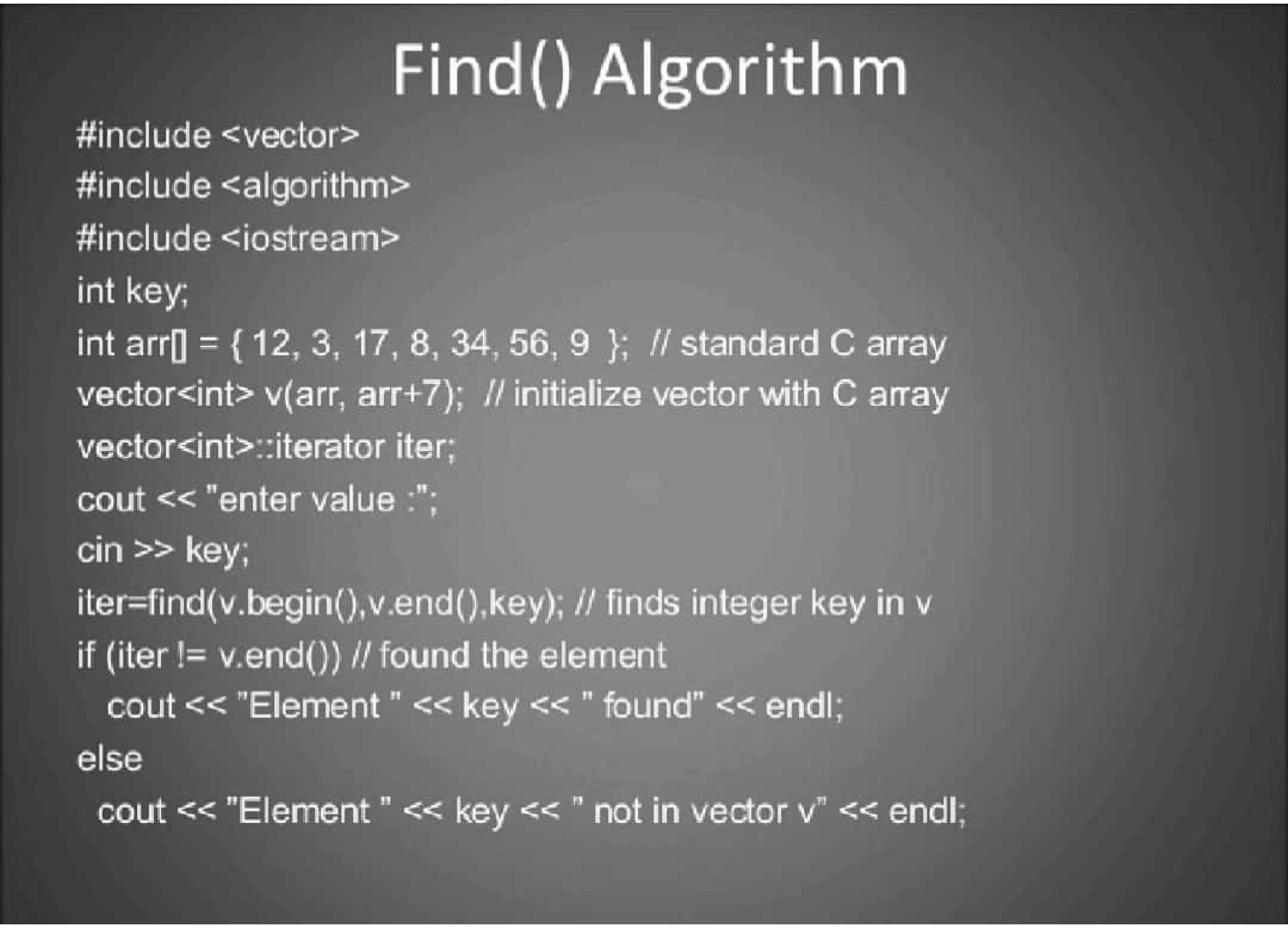
Scanned by CamScanner



For_Each() Algorithm

```
#include <vector>
#include <algorithm>
#include <iostream>
void show(int n)
{
  cout << n << " ";
}
 int arr[] = { 12, 3, 17, 8 };  // standard C array
vector<int> v(arr, arr+4);  // initialize vector with C array
for_each (v.begin(), v.end(), show); // apply function show
                // to each element of vector v
```

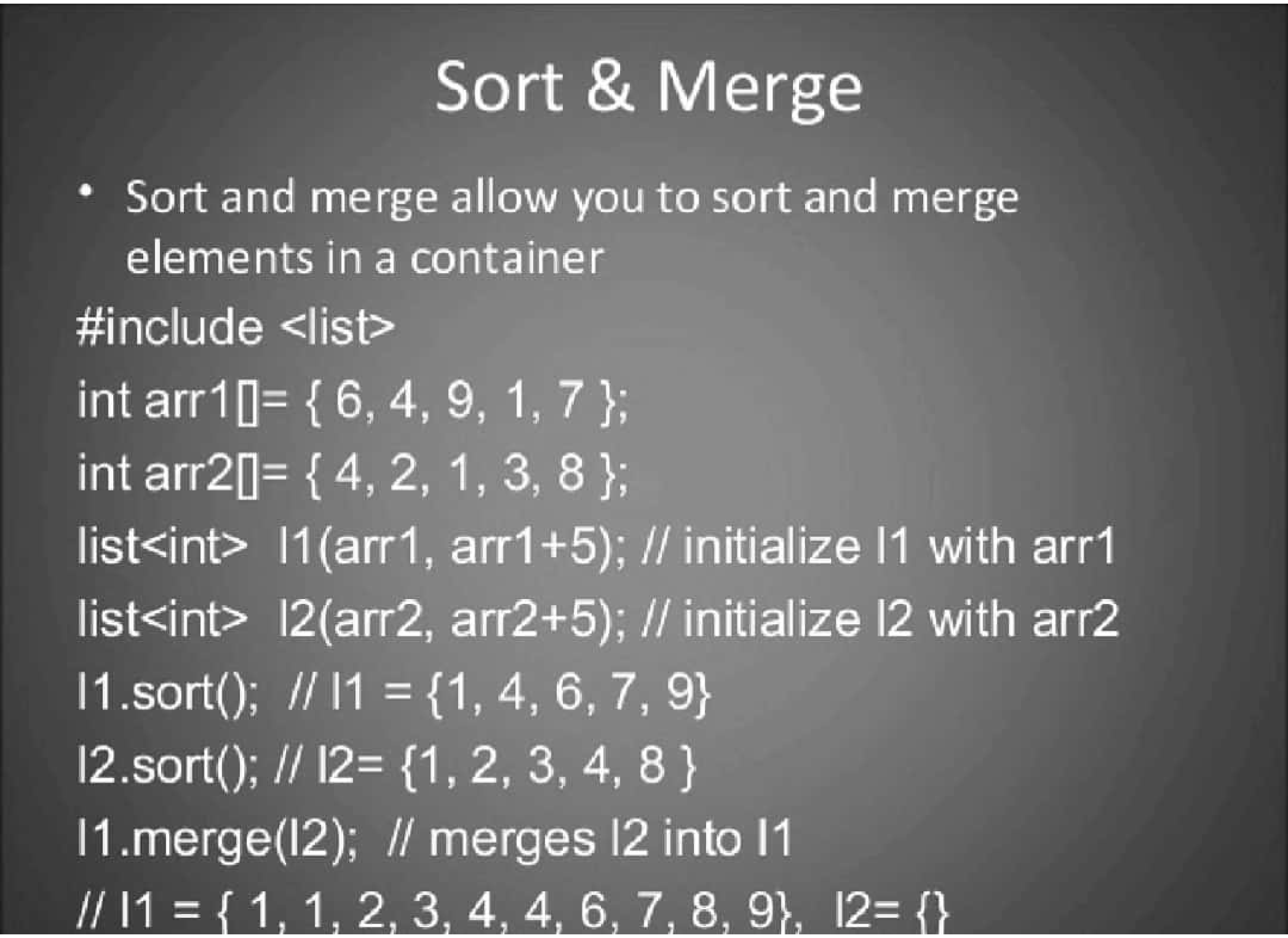
Scanned by CamScanner



Find() Algorithm

```
#include <vector>
#include <algorithm>
#include <iostream>
int key;
int arr[] = { 12, 3, 17, 8, 34, 56, 9  };  // standard C array
vector<int> v(arr, arr+7);  // initialize vector with C array
vector<int>::iterator iter;
cout << "enter value :";
cin >> key;
iter=find(v.begin(),v.end(),key); // finds integer key in v
if (iter != v.end()) // found the element
   cout << "Element " << key << " found" << endl;
else
  cout << "Element " << key << " not in vector v" << endl;
```

Scanned by CamScanner

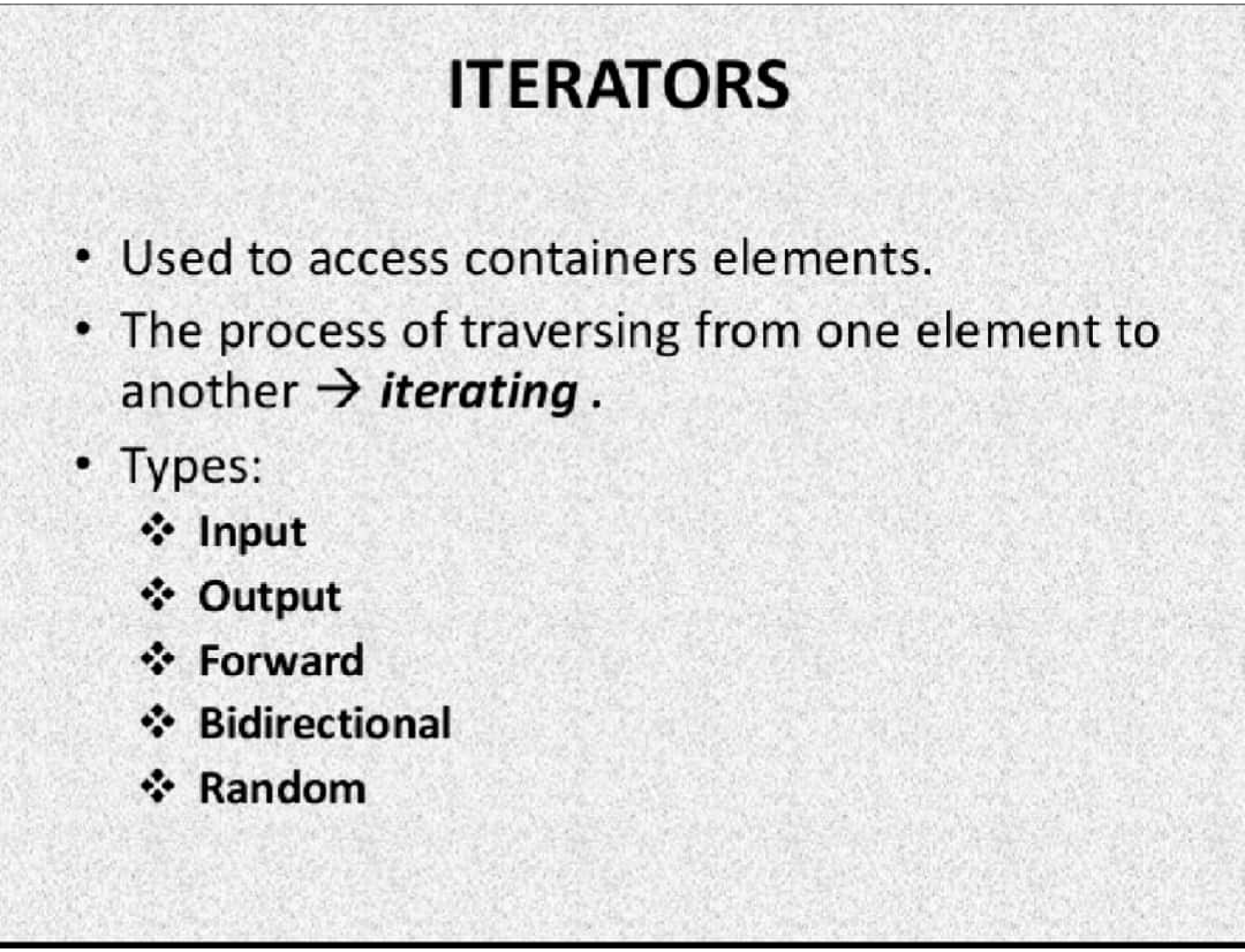


Sort & Merge

- Sort and merge allow you to sort and merge elements in a container

```
#include <list>
int arr1[]= { 6, 4, 9, 1, 7 };
int arr2[]= { 4, 2, 1, 3, 8 };
list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
list<int>  l2(arr2, arr2+5); // initialize l2 with arr2
l1.sort();  // l1 = {1, 4, 6, 7, 9}
l2.sort(); // l2= {1, 2, 3, 4, 8 }
l1.merge(l2);  // merges l2 into l1
// l1 = { 1, 1, 2, 3, 4, 4, 6, 7, 8, 9},  l2= {}
```

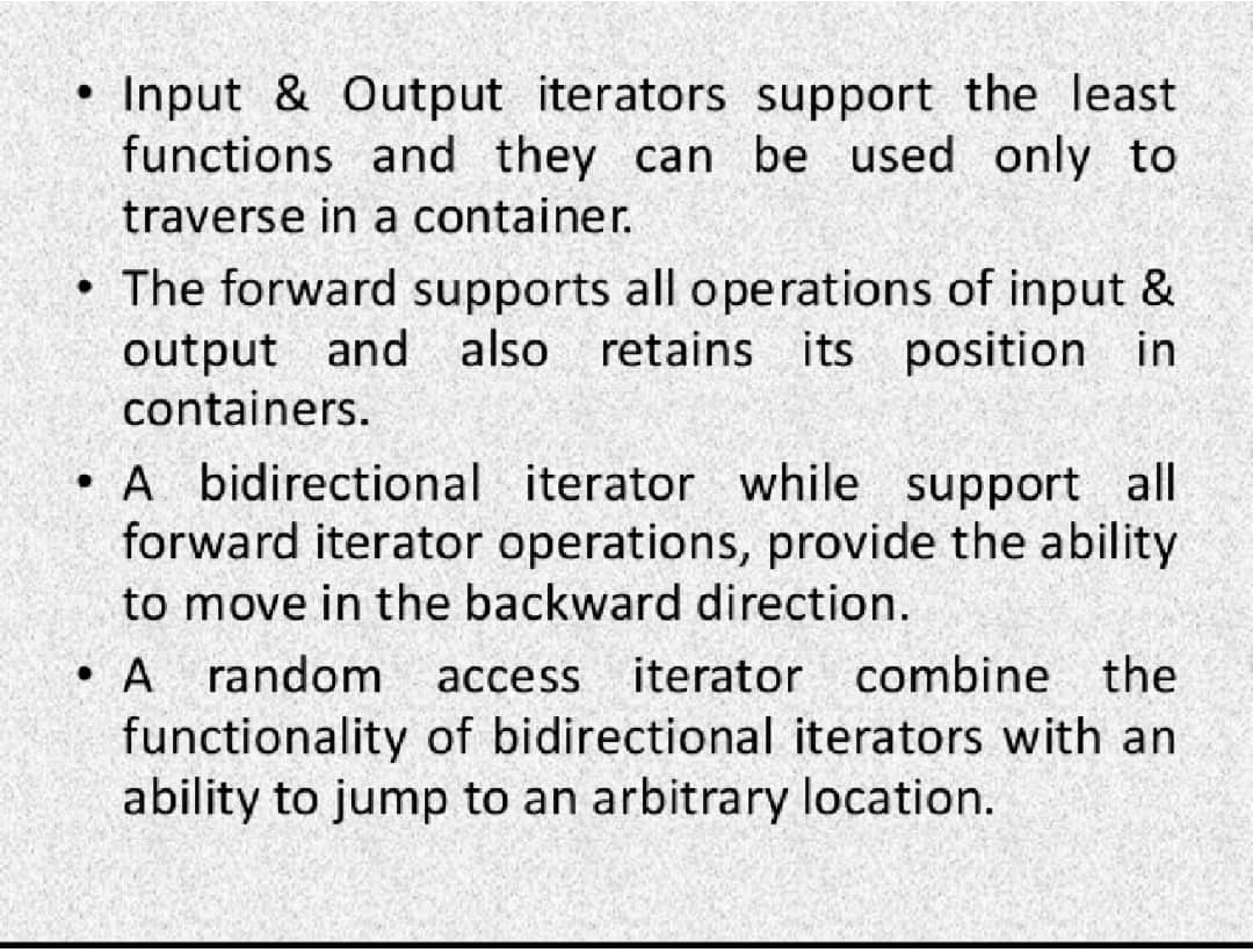
Scanned by CamScanner



ITERATORS

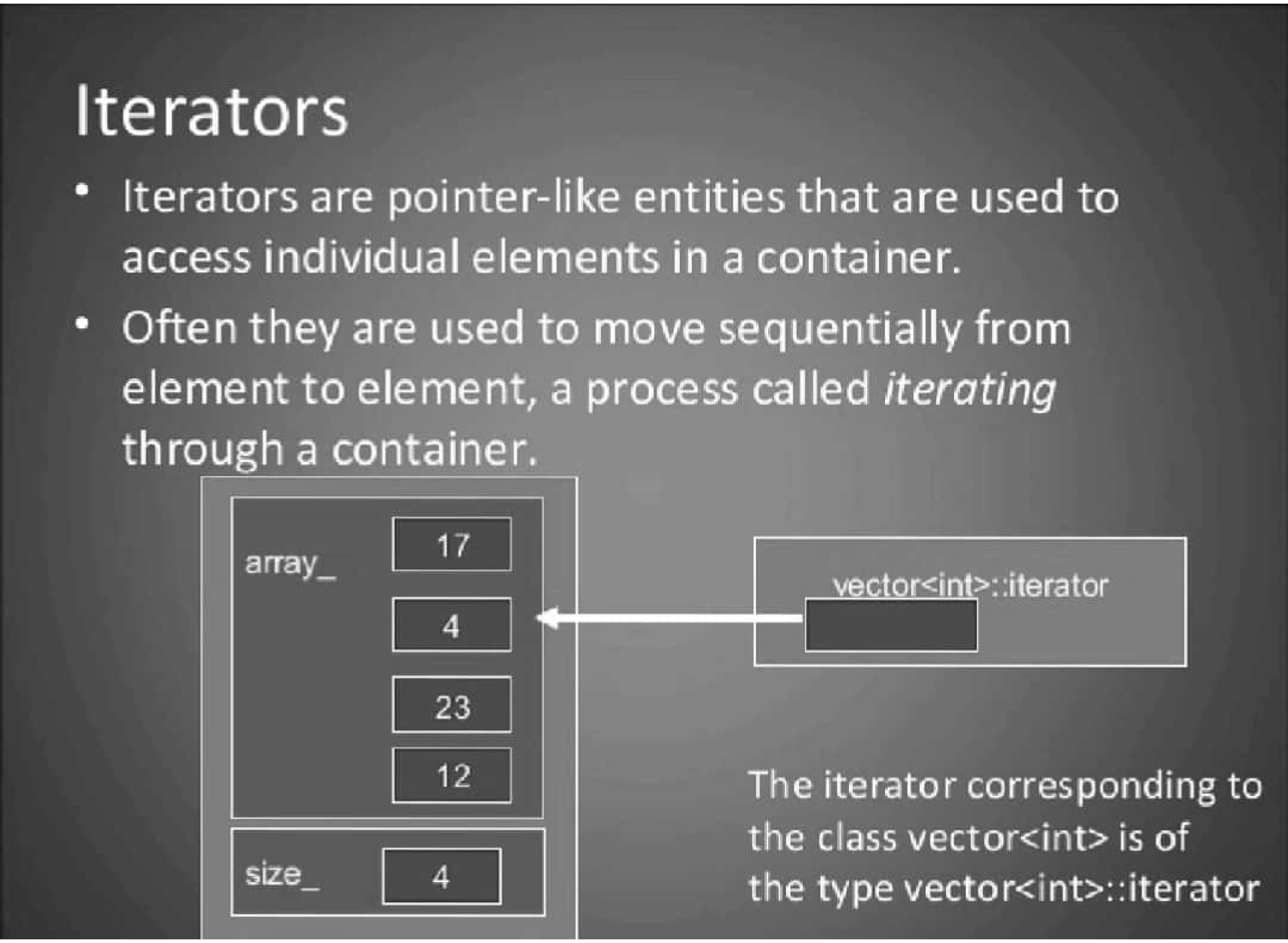
- Used to access containers elements.
- The process of traversing from one element to another → ***iterating*** **.**
- Types:
  - **Input**
  - **Output**
  - **Forward**
  - **Bidirectional**
  - **Random**

Scanned by CamScanner



- Input & Output iterators support the least functions and they can be used only to traverse in a container.
- The forward supports all operations of input & output and also retains its position in containers.
- A bidirectional iterator while support all forward iterator operations, provide the ability to move in the backward direction.
- A random access iterator combine the functionality of bidirectional iterators with an ability to jump to an arbitrary location.

Scanned by CamScanner

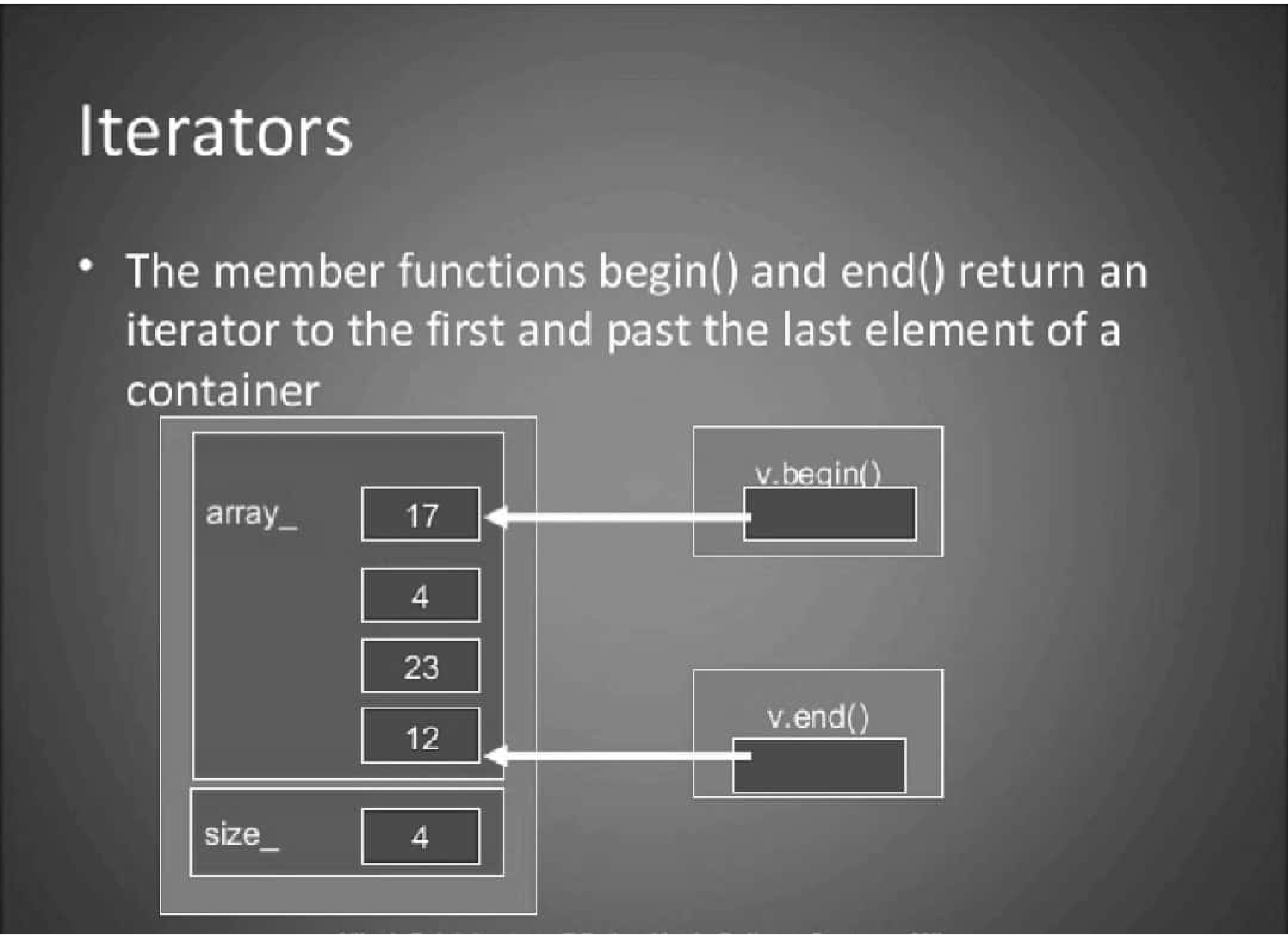


Iterators

- Iterators are pointer-like entities that are used to access individual elements in a container.
- Often they are used to move sequentially from element to element, a process called *iterating* through a container.

The iterator corresponding to the class vector<int> is of the type vector<int>::iterator

Scanned by CamScanner



Iterators

- The member functions begin() and end() return an iterator to the first and past the last element of a container

Scanned by CamScanner