C++ Standard Template Library

Iterators

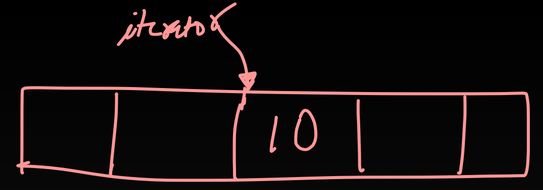
- Love Babbar

C++ Standard Template Library

Iterators

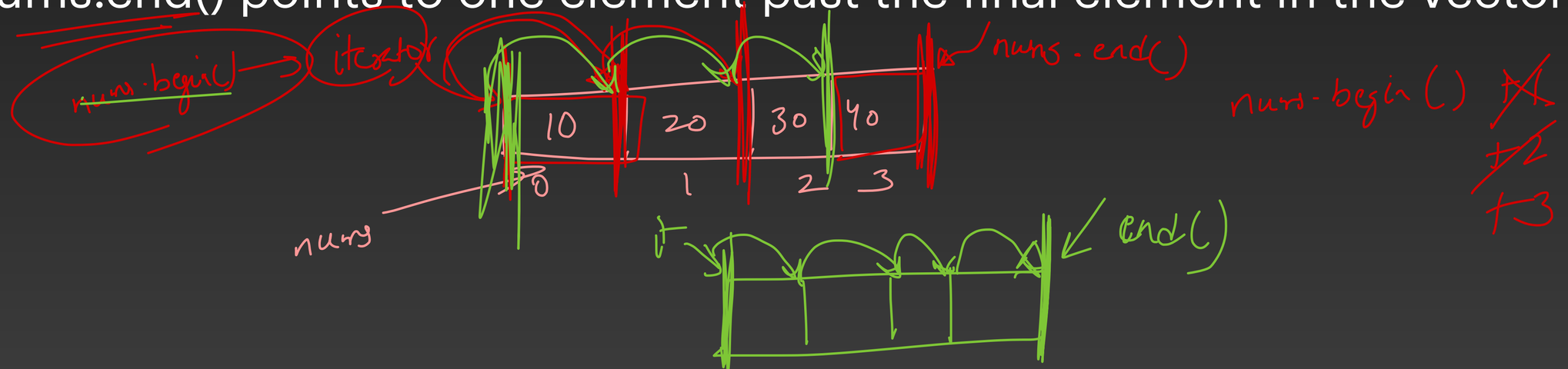
- Love Babbar

C++ Iterators



- An iterator is a pointer-like object representing an element's position in a container. It is used to iterate over elements in a container.

- Suppose we have a vector named nums of size 4. Then, begin() and end() are member functions that return iterators pointing to the beginning and end of the vector respectively.
  - nums.begin() points to the first element in the vector i.e 0th index
  - nums.begin() + i points to the element at the ith index.
  - nums.end() points to one element past the final element in the vector

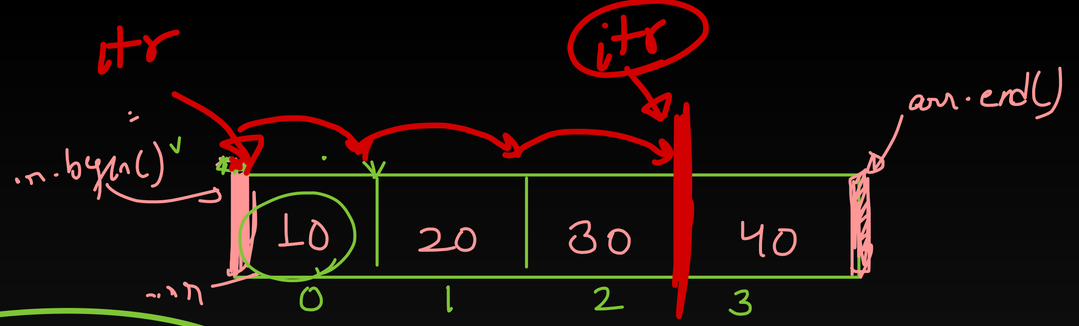


Creating & Traversing - Iterators

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

int main() {

    vector <string> languages = {"Python", "C++", "Java"};

    // create an iterator to a string vector
    vector<string>::iterator itr;

    // iterate over all elements
    for (itr = languages.begin(); itr != languages.end(); itr++) {
        cout << *itr << ", ";
    }

    return 0;
}
```

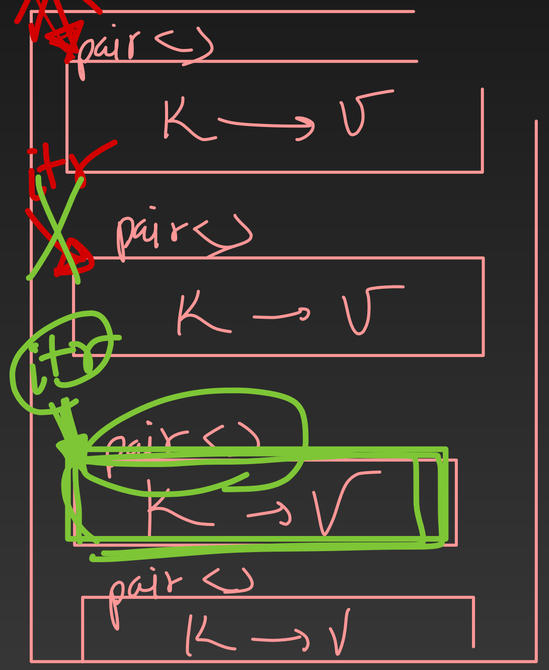
itr = itr1

Iterator Operations



itr → m
or
(*itr).m

itr++



Operations	Description
`*itr`	returns the element at the current position
`itr->m`	returns the member value `m` of the object pointed by the iterator and is equivalent to `(*itr).m`
`++itr`	moves iterator to the next position
`--itr`	moves iterator to the previous position
`itr + i`	moves iterator by `i` positions
`itr1 == itr2`	returns `true` if the positions pointed by the iterators are the same
`itr1 != itr2`	returns `true` if the positions pointed by the iterators are not the same
`itr = itr1`	assigns the position pointed by `itr1` to the `itr` iterator

(*itr).m

itr → m

itr → first

(*itr).first

itr → second

(*itr).second

iterator

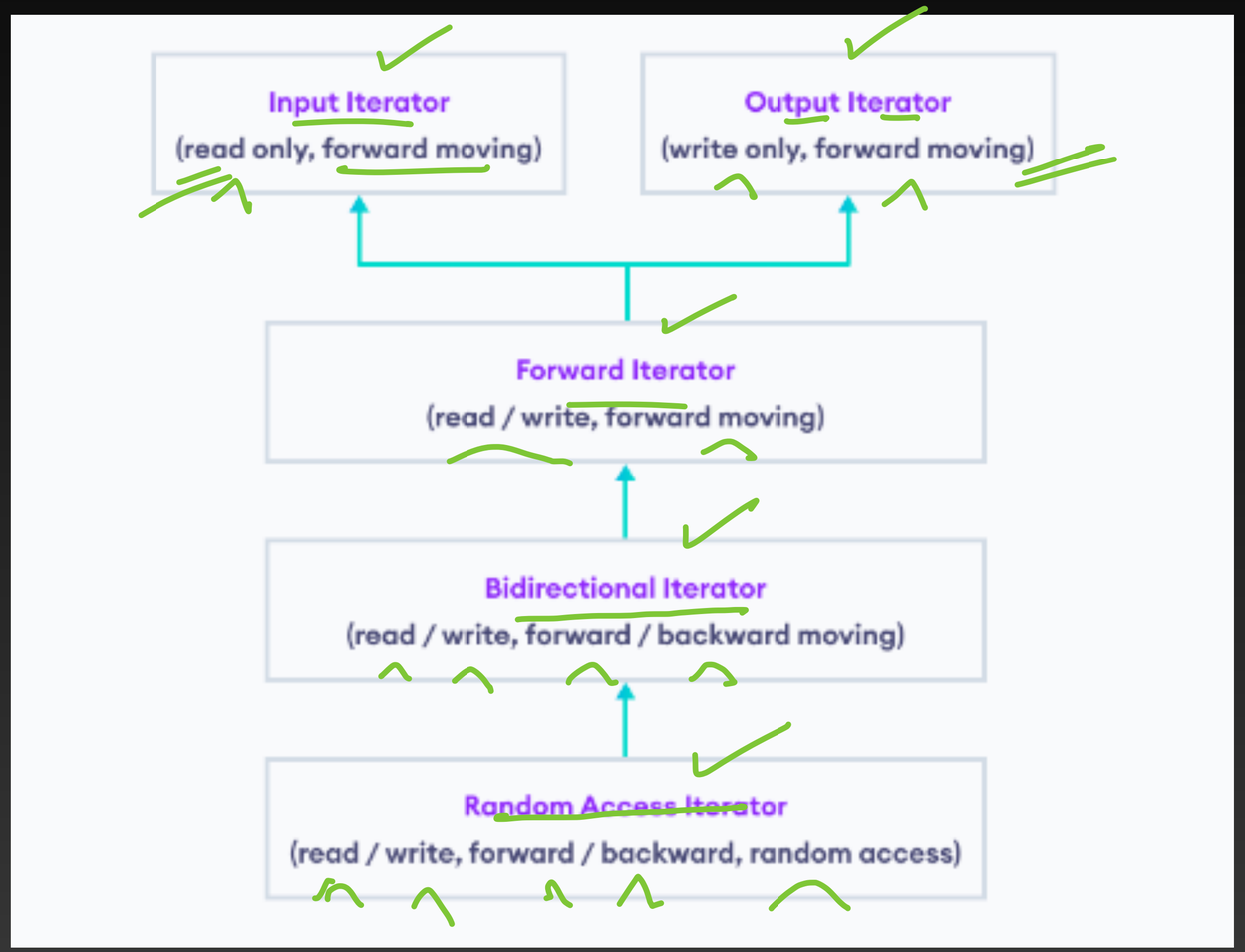
itr = arr.begin()

pos/location/address

*itr

Value at position where itr is pointing

Iterator Types



Input Iterator

read only → forward moving

- These iterators can only be used for reading values from a container in a forward direction. They are typically used for algorithms that need to read data from a container, such as std::find or std::for_each.

Ex

```
// create an input iterator to read values from cin
istream_iterator<int> input_itr(cin);
```

Output Iterator

write only → forward moving

- These iterators can only be used for writing values to a container in a forward direction. They are less commonly used compared to other iterator types.

Ex

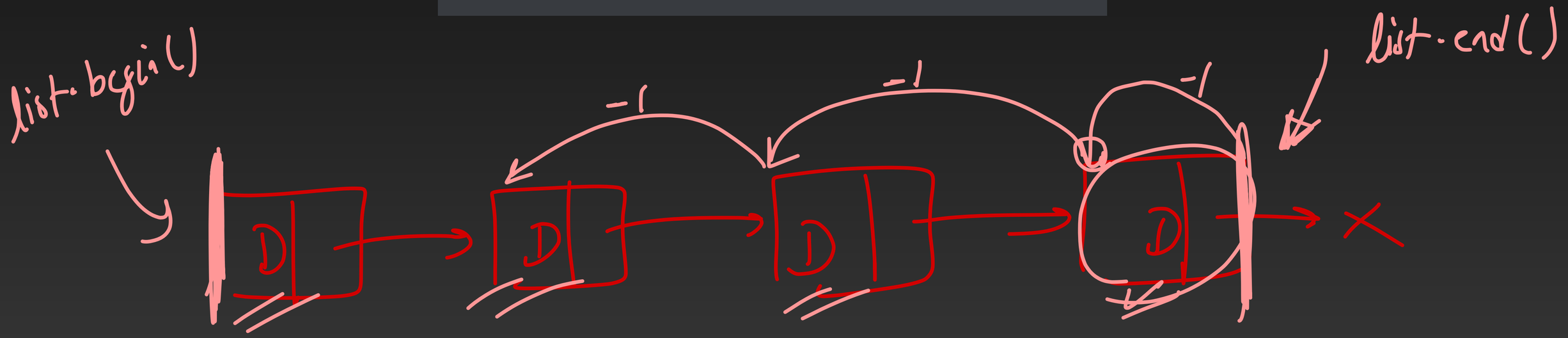
```
// create an output iterator to write integers to the console
ostream_iterator<int> output_itr(cout, " ");
```

Forward Iterator

read / write → forward dir^n

- These iterators combine the capabilities of both input and output iterators. They allow reading and writing values in a forward direction. Many container types, like lists, support forward iterators.

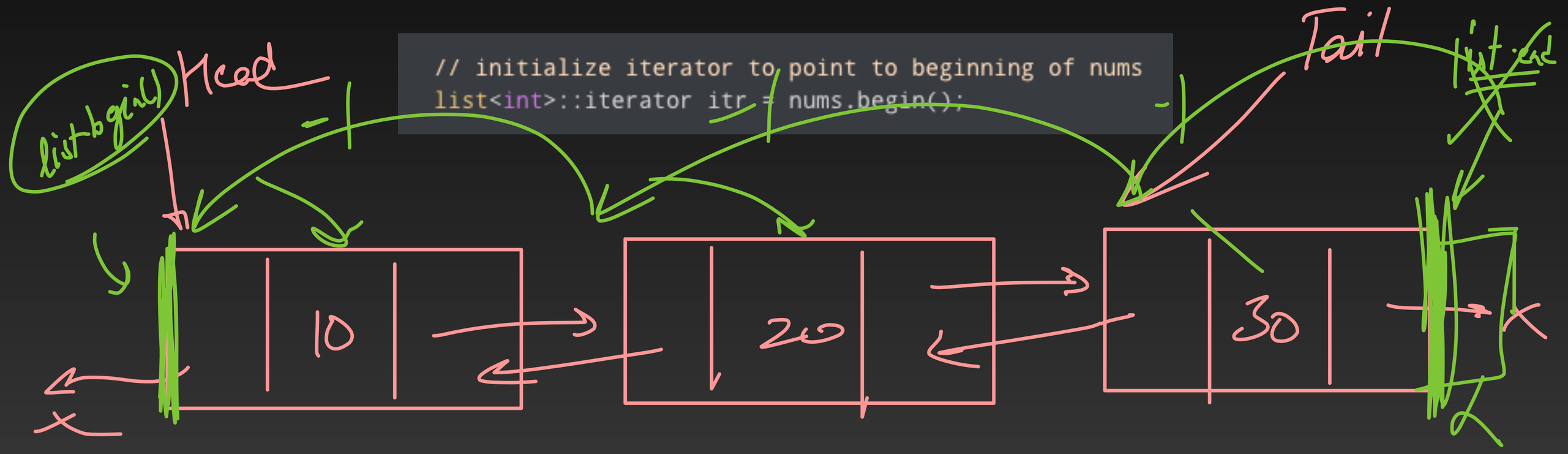
```
forward_list<int> nums{1, 2, 3, 4};

// initialize an iterator to point
// to beginning  of a forward list
forward_list<int>::iterator itr = nums.begin();
```



Bi-directional Iterator

- These iterators can move both forward and backward within a container. They are supported by containers like lists and double-ended queues (deques).

```
// initialize iterator to point to beginning of nums
list<int>::iterator itr = nums.begin();
```



Random Access Iterator

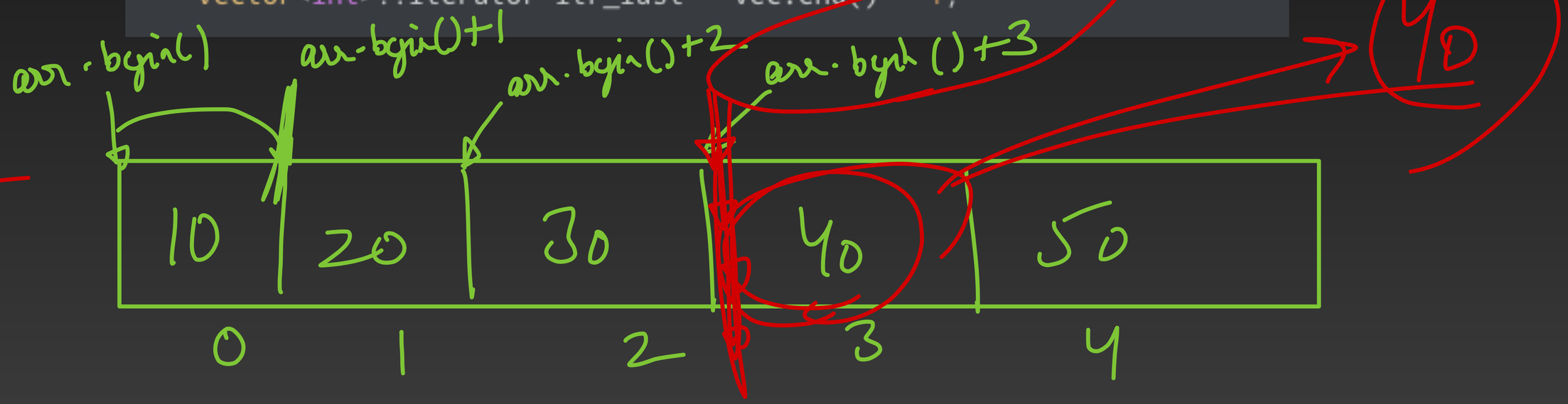
- These iterators offer full navigation capabilities, allowing you to move to any element within a container in constant time. Vectors, arrays, and deques provide random access iterators

read / write / forward / Backward / Random Access

it => arr.begin() + 3

```
// create iterators to point to the first and the last elements
vector<int>::iterator itr_first = vec.begin();
vector<int>::iterator itr_last = vec.end() - 1;
```

*it



Operations supported by Iterators

Iterator Type	Supported Operators
Input Iterator	`++`, `*`, `->`, `==`, `!=`
Output Iterator	`++`, `*`, `=`
Forward Iterator	`++`, `*`, `->`, `==`, `!=`
Bidirectional Iterator	`++`, `--`, `*`, `->`, `==`, `!=`
Random Access Iterator	`++`, `--`, `*`, `->`, `[]`, `+`, `-`, `<`, `<=`, `>`, `>=`, `==`, `!=`

Why use Iterators ?

- Working with Algorithms: C++ has many ready-to-use algorithms like finding elements, sorting, and summing values. Iterators help you apply these algorithms to different types of data containers like arrays or lists.
- Saving Memory: Instead of loading a huge set of data all at once, iterators let you deal with one item at a time, which saves memory.
- Uniform Approach: Iterators allow you to interact with different kinds of data containers (like vectors or sets) in the same way. This makes your code more consistent and easier to manage.
- Simpler Code: By using iterators, a lot of the repetitive details of going through data are taken care of, making your code cleaner and easier to read.

Why use Iterators ?

- Working with Algorithms: C++ has many ready-to-use algorithms like finding elements, sorting, and summing values. Iterators help you apply these algorithms to different types of data containers like arrays or lists.
- Saving Memory: Instead of loading a huge set of data all at once, iterators let you deal with one item at a time, which saves memory.
- Uniform Approach: Iterators allow you to interact with different kinds of data containers (like vectors or sets) in the same way. This makes your code more consistent and easier to manage.
- Simpler Code: By using iterators, a lot of the repetitive details of going through data are taken care of, making your code cleaner and easier to read.

Why use Iterators ?

- Working with Algorithms: C++ has many ready-to-use algorithms like finding elements, sorting, and summing values. Iterators help you apply these algorithms to different types of data containers like arrays or lists.
- Saving Memory: Instead of loading a huge set of data all at once, iterators let you deal with one item at a time, which saves memory.
- Uniform Approach: Iterators allow you to interact with different kinds of data containers (like vectors or sets) in the same way. This makes your code more consistent and easier to manage.
- Simpler Code: By using iterators, a lot of the repetitive details of going through data are taken care of, making your code cleaner and easier to read.