Pairs

* Pros :-

1. They are used to store element in 'pairs' of any 2 data types (sim or diff form each other).
2. Utility : to perform operation of data types that are supposed to be mapped(contextually) in one go.
3. It is often used in situations where you need to return two values from a function or when you want to store related data together.

* Cons:-

1. is limited to storing only two elements. If you need to store more than two related values, you might need to use std::tuple or define a custom structure or class.
2. Unlike structs or classes where you can provide member functions and encapsulate behavior, std::pair is a simple aggregate with no functionality beyond holding two values. This means you cannot enforce invariants or provide specialized behavior.

Vectors

* Imp Pts:-

1. The time complexity of size() for a std::vector in C++ and len() for a list in Python is O(1) because the size or length of these data structures is stored as an attribute that can be directly accessed without iteration.
2. Local vector = O(10^5) and

Global vector = O(10^7); size limits sim to arr.

1. vector <int> v1= v2 ; creates ‘deep’ copy not ref as in case of python.
2. Vectors are ana to lists in python.
3. list() ana in c++:-

string str = "abc";

vec(str.begin(), str.end());

* Some func in Vectors (ana to list):-

1. push\_back() : vec.push\_back(10); // Adds 10 to the end of the vector
2. pop\_back() : vec.pop\_back(); // Removes the last element (3) from the vector
3. size() : vec.size() //Returns the number of elements in the vector.nsert()
4. insert() : vec.insert(vec.begin() + 1, 10); // Inserts 10 at index 1
5. erase() : vec.erase(vec.begin() + 2); // Removes element at index 2 (value: 3)
6. clear() : vec.clear(); // Removes all elements from the vector
7. begin() and end(): Return iterators pointing to the first and one past the last element of the vector, respectively.

find() : Searches for an element in the vector and returns an iterator to its first occurrence. Ex: find(vec.begin(), vec.end(), 3);

Iterators

* Imp Pts:-

1. They are pointer like structures.
2. it++ : goes to next iterator

it +1 : goes to next location; will lead to ‘seg fault or can lead to mem leak’ in case of unordered data str.

1. (\*it).first ⬄ it ->first

Maps

* Imp Pts: -

There three maps viz,

* Ordered
* Unordered
* Mutlimap

1. Ordered Map(or simply Map)
2. It is implemented by “Red Black Tree” Data str; a self balancing tree like the AVL tree.
3. Hence time for ins, acess in O(logn)
4. All elem are stored in sorted order, by making comp with the parent node of Red Black Trees.
5. Hence in the case of “Strings” as key value, ins = <string>.size()\*O(logn).
6. Unordered Map
7. All pts same as abv; apart form the fact that it takes “hashable” obj as it’s first argument.(i.e, no pairs, for ex)
8. Hence all O(logn) convered to O(1) operations(avg case).
9. MultiMaps
10. For sim to maps in the implementation; only fact that it can store duplicate keys.
11. Moreover, elem are added to mutimaps using <multimaps>.insert(<keys>,<values>) and not by using “[]” op.

Sets

* Imp Pts:-

All the concepts ana wrt Maps; only diff lies that it stores keys and not key value pairs.