Lists

Type
Collection of values such as:
- boolean = True/False
- integer = (..., -1, 0, 1, …)
- float,
- etc

Processor has the knowledge of these and knows what they are.

Data Type:
means the type of data itself and the operations we can perform to manipulate that type.

Meaning saying integer datatype we mean the integer type itself and all operations: *+==

Abstract Data Type:
Processor does not know what they are.
eg: list, stack, queue, dictionary, set, etc.

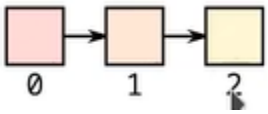
They only define a datatype (document how a datatype should behave) but doesn't say anything about the implementation. The implementation is the realm of the data structure

Data Structure:
Implementation of an ADT
eq: Array, singly linked list, doubly linked list, hash table, trees, etc.

lists are an ADT but can be implemented by different data structures such as Array, singly linked list, etc.

List ADT

Collection of items. It is an ORDERED collection of items



what kind of operation do we want to be able to perform on this ADT?

Operations

- Typical set of operations (*API*)

```
// Return number of items in list
unsigned int Size();
// Return item at position @pos
const T& Get(const unsigned int pos);
// Return position of first occurence of @item (-1 if not found)
int Find(const T &item);
// Remove item at position @pos
void Remove(const unsigned int pos);
// Insert @item at position @pos
void Insert(const T &item, const unsigned int pos);
```

Fixed array implementation

```
template <typename T>
class ListFixedArray {
  private:
    // fixed capacity of 3 items
    static constexpr int capacity = 3;
    std::array<T, capacity> items;
    unsigned int curr_size = 0;
    // ...
};
```

-we have a predefined capacity for the fixed array (std::array)
-if capacity is a variable, and the compiler needs to know the value of this variable, the compiler needs to be a constant (needs to be known by the compiler at the moment the class loads).
That is why we used static constexpr, now capacity is hardcoded.

- `static` to make `capacity` exist without instantiating and `constexpr` to make it available to compiler
- `std::array` is equivalent to a C-style array (`T items[capacity]`)
- Member variables already initialized

```
public:
  ListFixedArray() = default;
  ~ListFixedArray() = default;
```

- `default` to have the compiler generate the corresponding constructor or destructor:
  - No initialization list
  - Empty compound statement

List API

Size

```
    unsigned int size(){
        return curr_size;
    }
```

We will always maintain the curr_size variable so that the user can get the size.

Get

```
    const T& Get(const unsigned int pos){
        if(pos >= curr_size)
            throw std::out_of_range("Position out of range!");
        return items[pos];
    }
```

the user will give us a certain position in the list and request the item at that position.

✅ Returning by **reference** (`T&`) means:

- You **don't make a copy** of the object.
- You're giving the caller a **reference to the actual item** stored in the array.
- This is faster, especially if `T` is a large object (like a `std::string`, `std::vector`, or a custom class).

🚫 If you return by **value** (`T`):

- The entire object is **copied**, which can be expensive.
- The caller gets a **new copy**, not the original item in your array.

✅ `const` reference (`const T&`)

- Prevents the caller from modifying the returned item.
- Safer when you're just "reading" the item.

> 💡 If you want the caller to be able to **modify** the item (e.g. `arr.Get(0) = 5;`), then drop the `const`.

mini refresher on references
int a = 5;
int& ref = a; // ref is a reference to a

- `ref` is another **name** for `a`
- You don't need to use `*` to dereference or `&` to take the address
- Any change to `ref` changes `a`

✅ When you pass by reference:

cpp Copy Edit

```cpp
void foo(int& x) {
    x = 10;
}
```

Calling it:

cpp Copy Edit

```cpp
int a = 3;
foo(a);  // now a == 10
```

You're passing **a reference**, but **you never use** `*` **or** `&` **inside the function.**

> Think of it like an alias — it's like you're directly working with the original variable, but under a different name.

notice above when calling foo, we did not do: foo(&a)

📌 Pointers are more manual:

cpp Copy Edit

```cpp
void bar(int* x) {
    *x = 20; // manual dereference
}
```

Calling it:

cpp Copy Edit

```cpp
int b = 5;
bar(&b);  // pass address manually
```

Find

```cpp
int Find(const T &item){
    for (unsigned int i = 0; i < curr_size; i++){
        if (items[i] == item){
            return i;
        }
    }
    return -1;
}
```

❓ Why `const T& item` in `Find(...)`?

cpp Copy Edit

```cpp
int Find(const T &item)
```

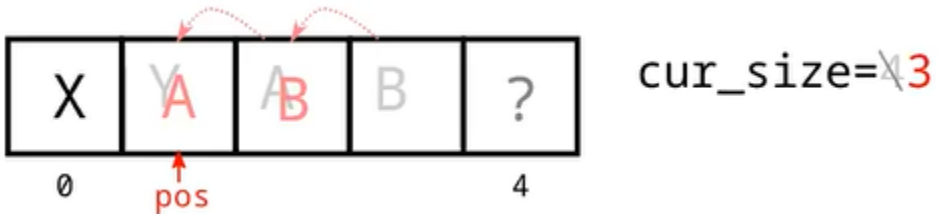
✅ `const T&` means:

1. **We're passing the argument by reference** → avoids copying.
   - If `T` is something big like `std::string`, `std::vector`, or a custom object, copying it would be expensive.
   - Passing by reference is fast — it's just passing the **address**.
2. **We're not going to modify the item** inside the function.
   - Adding `const` makes it clear and safe.

Remove

```
    void Remove(const unsigned int pos){
        if (pos >= curr_size){
            throw std::out_of_range("Position out of range!");
        }

        for (auto i = pos + 1; i < curr_size; i++){
            items[i - 1] = items[i];
        }
        curr_size--;
    }
```



2 scenarios:
scenario 1 - removing the last item in the list: |X|Y|A|B| curr_size = 4
user calls Remove(3) (he wants to remove item 3). In that case we don't have to do much and just remove the item at the end and update the curr_size to 3

scenario 2 - remove is being called to remove an item in the middle of the list:
|X|Y|A|B| curr_size = 4
we remove the item and shift all the items after that item one position to the left.

Insert

```
    void Insert(const T &item, const unsigned int pos){
        if (pos > curr_size){
            throw std::out_of_range("Position out of
range!");
        }
        if (curr_size == capacity){
            throw std::overflow_error("List full!");
        }
```

```
        if (pos != curr_size){
            // Move existing items(s) if insertion before
them
             auto i = curr_size - 1;
             do {
                 items[i + 1] = items[i];
             } while (i-- != pos);
        }
        items[pos] = item;
        curr_size++;
    }
```