Vectors

Vectors are described as being **very array-like** in appearance and visualisation, using different blocks to store data and having associated indices, just like arrays.

The key difference highlighted between Arrays and Vectors is that while Arrays have a **fixed size**, Vectors are **dynamic in nature**, meaning their size can change. This solves the problem encountered with fixed-size arrays where, for example, an array created for 100 students wouldn't be usable if the number of students increased to 1000. Vectors can dynamically resize to accommodate more elements.

The implementation of Vectors comes from the **Standard Template Library (STL)** in C++. The STL is described as a library or "toolbox" that contains the pre-written code for implementing many data structures like stacks, hash tables, and queues. Using the STL is standard practice in coding tests and interviews because implementing these data structures from scratch for every problem would be too time-consuming. Vectors are presented as one of the tools available within the STL. Data structures implemented in the STL, like Vectors, Queues, Stacks, and Sets, are called **STL Containers** because they are designed to store data.

To use Vectors in C++, it is necessary to include the `<vector>` **header file**. The material advises against using the `<bits/stdc++.h>` header file often seen on online platforms because knowing the specific header file for each data structure is important for learning and for cleaner code in professional settings, avoiding potential namespace conflicts or including unnecessary components.

Creating (Declaring) and Initialising Vectors: Several methods are discussed for creating Vectors:

1.**Default Construction:** `vector<type> name;`. This creates an **empty vector** with a size of zero. Attempting to access an index (like index 0) on an empty vector will result in a **segmentation fault**.

2.**Initialisation with elements:** `vector<type> name = {element1, element2, ...};`. Similar to array initialisation, this creates a vector whose size is determined by the number of elements provided.

3.**Initialisation with fixed size and default value:** `vector<type> name(size, value);`. This creates a vector of a specified `size`, where every element is initialised to the provided `value`. If only the size is provided (`vector<type> name(size);`), elements might be default-initialised (e.g., to 0 for integers).

When compiling code that uses Vectors, especially with newer C++ standards, it might be necessary to specify the standard using a flag like `-std=c++11` during compilation.

Accessing Vector Elements: Vector elements are accessed using **zero-based indexing**, just like arrays.

•`vector_name[index]` allows accessing or modifying the element at the specified index.
•`vector_name.at(index)` is an alternative way to access an element at a specific index. Accessing an index outside the valid range (0 to size - 1) will lead to an error, potentially a segmentation fault.

Looping through Vectors: The **For-Each loop** is a commonly used special type of loop for iterating through vectors and other STL containers. The syntax is `for (type value : vector_name) { /* use value */ }`, where `value` is an iterator variable that directly holds the **value** of the element at each index, rather than the index itself. The `type` of the iterator should match the vector's element type.

Common Vector Functions: Vectors come with associated functions to perform operations:
•`size()`: Returns the current number of elements in the vector.
•`push_back(element)`: Adds an `element` to the **end** of the vector. This increases the vector's size. Elements are added in the order they are pushed.
•`pop_back()`: **Removes the last element** from the vector. It automatically removes the element at the last index, so no value needs to be specified. This decreases the vector's size.
•`front()`: Returns a reference to the **first element**.
•`back()`: Returns a reference to the **last element**.
•`at(index)`: Returns a reference to the element at the specified `index`, providing bounds checking.

Static vs. Dynamic Memory Allocation and Vector Implementation:

•**Static memory allocation** happens at **compile time** (the stage where the compiler checks code before execution). This is typically where Arrays are allocated, specifically in the **Stack memory**. The size must be fixed and known at compile time.

•**Dynamic memory allocation** happens at **run time** or execution time. This is where Vectors get their memory allocated, specifically in the **Heap memory**. Because allocation happens at run time, the size can change dynamically. This dynamic allocation is why vectors can resize.

Internally, a Vector is implemented using an **Array** in memory. When a vector is initially created, it might have a small internal array or none at all. When the first element is pushed back, an internal array is created. If `push_back` is called again and the current internal array is full, the vector performs an automatic operation: it **creates a *new* internal array with double the capacity** of the old one, copies all the existing elements from the old array to the new one, adds the new element, and then deletes the old array from memory. This doubling of capacity happens automatically and gives the user the impression that the vector is magically growing.

Vectors have two relevant properties related to their size and memory:

•**Size**: The actual number of elements currently stored in the vector.

•**Capacity**: The total number of elements the underlying internal array can currently hold without needing to allocate a new, larger array. Capacity is always greater than or equal to size. The capacity doubles when the current space is exhausted by a `push_back` operation.

Pass by Reference with Vectors in Functions: When Vectors (or other C++ containers) are passed as arguments to functions, they are **passed by value by default**. This means a copy of the vector is made, and changes inside the function would not affect the original vector in the calling code (like `main`). To allow changes inside the function to affect the original vector, they must be **passed by reference**. This is achieved by adding an **ampersand (`&`)** before the vector's name in the function's parameter list (e.g., `void myFunction(vector<int>& nums)`). Passing by reference creates an "alias" or alternate name for the original vector, so operations on the alias directly modify the original vector.