Introduction to Vectors

- **What is a Vector in C++?**
  - A vector is a dynamic array______________ that can grow or shrink in size as needed.
  - Part of the Standard Template Library (STL) in C++.
  - Offers more flexibility__________ and functionality_________ compared to traditional arrays.
- **Key Features of Vectors:**
  - Dynamic Size____________**:** Automatically adjusts its size.
  - Direct Element Access______**:** Provides random access to elements.
  - Memory Efficient__________**:** Allocates memory as needed, reducing waste.
  - Flexible________________**:** Supports various operations like insertion, deletion, sorting, and searching.

Why Use Vectors?

- Ease of Use____________**:** Simplifies memory management compared to traditional arrays.
- Safety________________**:** Offers bounds checking with methods like **at()**.
- Versatility______________**:** Ideal for scenarios where the size of the data set is not known beforehand or changes dynamically.
- Compatibilty with STL Algorithms**:** Seamlessly works with numerous algorithms provided by the STL for sorting, searching, etc.

Vectors vs Arrays

- **Introduction:**
  - Understanding the difference between vectors and traditional arrays is crucial in C++ programming.
  - Both are used to store sequences of elements__________, but they have significant differences.
- **Arrays:**
  - Basic data structure in C++____.
  - Fixed______ size - the size needs to be known at compile-time.
  - Example declaration: **int MyArray[10];**
- **Vectors:**
  - Part of the Standard Template Library (STL).
  - Dynamic_____ size - can grow or shrink at runtime.
  - Example declaration: **std::vector<int> MyVector;**
-

- **Key Differences:**
  - Size Flexibility**:** Vectors can change size during runtime, arrays cannot.
  - Memory Management**:** Vectors handle memory automatically, while arrays require manual management.
  - Safet**:** Vectors provide more safety with functions like **at()** which checks bounds.
  - Functionality**:** Vectors come with a lot of built-in functions like **insert()**, **erase()**, **push_back()**, etc., which are not available with arrays.
- **Performance:**
  - Arrays may offer slightly better performance due to their static nature, especially for small and fixed-size data.
  - Vectors have an overhead for dynamic memory and additional functionalities
- **Use Cases:**
  - **Arrays:** When the number of elements is known and fixed. For example, storing the days of the week.
  - **Vectors:** When the data is dynamic or the size might change. For example, storing a list of user inputs.

Declaring and Initializing Vectors

- **Declaring a Vector:**
  - Basic declaration: std::vector<int>myVector**;**

```cpp
#include <vector>                  // Include the vector header file
int main() {
    std::vector<int> myVector;     // Declare a vector of integers
    return 0;
}
```

- **Initializing a Vector:**
  - **Initialization with Size and Value:**
    - Creates a vector with a specified size,
all elements have the same specified value.
    - **std::vector<int> myVector3(4, 100); // Vector of 4 integers, each initialized to 100**
  - **Initializer List:**
    - Creates a vector and initializes it with a list of values.
    - **std::vector<int> myVector4 = {10, 20, 30, 40};**
  - **From Another Vector:**
    - Creates a copy of another vector.
    - **std::vector<int> myVector5 = myVector4;   // A copy of v4**
- **Initializing a Vector:**
  - **Default Initialization:**
    - Creates an empty vector.
    - **std::vector<int> myVector1;**
  - **Initialization with Size:**
    - Creates a vector with a specified size, default-initialized elements.
    - **std::vector<int> myVector2(5); // Vector of 5 integers**

- **Vector of Objects:**
  - Vectors can store not just primitives but also objects.
  - **std::vector<MyClass> objVector;**
- **Special Member Functions:**
  - Copy Constructor: **std::vector<int> vec(copyVec);**
  - Move Constructor: **std::vector<int> vec(std::move(sourceVec));**

Accessing Elements

- **Direct Access:**
  - operator[]: Provides direct access to the specified element.
    - Fast but does not perform bounds checking.
    - Example: **int value = myVector[2];**
  - at(): Provides direct access with bounds checking.
    - Throws **std::out_of_range** exception if index is out of bounds.
    - Example: **int value = myVector.at(2);**
- **Accessing First and Last Elements:**
  - front(): Access the first element.
    - Example: **int firstValue = myVector.front();**
  - back(): Access the last element.
    - Example: **int lastValue = myVector.back();**
- **Iterators:**
  - Provide a way to access elements in a range-based manner.
  - Begin and end iterators: begin**()** and end**()**.

```
for (auto it = myVector.begin(); it != myVector.end(); ++it) {
    std::cout << *it << ' ';
}
```

- **Range-based For Loop:**

```
for (int value : myVector) {
    std::cout << value << ' ';
}
```

Vector Iterators

- Iterators are a key concept in C++ STL, providing a way to access elements of a vector sequentially.
- **What are Iterators?**
  - Iterators are objects that point to elements in a container like vectors.
  - They act similarly to pointers but are more flexible and safer.
- **Types of Iterators:**
  - **Regular Iterator (iterator):** Allows reading and writing of vector elements.
  - **Constant Iterator (const_iterator):** Only allows reading, not modification.

- **Accessing Iterators:**
  - **Begin and End Iterators:**
    - begin______(): Returns an iterator to the start of the vector.
    - end_____(): Returns an iterator to the end (one past the last element) of the vector.

```
std::vector<int> myVector = {1, 2, 3, 4, 5};
for (auto it = myVector.begin(); it != myVector.end(); ++it) {
    std::cout << *it << " ";
}
```

- **Accessing Iterators:**
  - **Constant Iterators:**

```
std::vector<int> MyVector = {10, 20, 30, 40, 50};

// Declare a const_iterator
std::vector<int>::const_iterator cit;

std::cout << "Vector elements using const_iterator: ";
for (cit = MyVector.cbegin(); cit != MyVector.cend(); ++cit) {
    std::cout << *cit << " ";
    // Note: *cit = 5; // This would be an error,
    // as const_iterator doesn't allow modification
}
std::cout << std::endl;
```

- **Using Iterators in Algorithms:**
  - Iterators are widely used in STL algorithms, such as **std::sort(vec.begin(), vec.end());**
- **Safety and Best Practices:**
  - Be careful with iterator invalidation - adding or removing elements in a vector can invalidate iterators___..
  - Prefer **const_iterator** when modification of elements is not needed for added safety.

Modifying Vectors

- **Adding Elements:**
  - push_back_____(): Adds an element to the end of the vector.
    - Example: **vec.push_back(100); // Adds 100 to the end of vec**
  - emplace_back________(): Constructs an element in-place at the end, often more efficient than **push_back**.
    - Example: **vec.emplace_back(100);**

```
std::vector<int> myVector;
myVector.push_back(10);  // Adds 10 at the end
myVector.emplace_back(20); // Constructs an integer with value 20 at the end
```

- **Inserting Elements:**
  - insert**():** Inserts elements at a specified position or range.
  - **vec.insert(vec.begin() + 2, 300);**

```
// Inserts 30 at the second position
myVector.insert(myVector.begin() + 1, 30);

// Inserts three times 40 at the fourth position
myVector.insert(myVector.begin() + 3, 3, 40);
```

- **Removing Elements:**
  - pop_back**():** Removes the last element.
    - Example: **vec.pop_back();**
  - erase**():** Removes elements at a specified position or range.
    - Example: **vec.erase(vec.begin() + 1);**

```
myVector.pop_back(); // Removes the last element

// Erases the second element
myVector.erase(myVector.begin() + 1);

// Erases a range of elements, from the second to the fourth elemen
myVector.erase(myVector.begin() + 1, myVector.begin() + 4);
```

- **Resizing and Clearing:**
  - resize**():** Changes the size of the vector, adding default elements or trimming the size as necessary.
    - Example: **vec.resize(10);**
  - clear**():** Removes all elements from the vector.
    - Example: **vec.clear();**

```
// Resizes vec to 5 elements. If size is greater, truncates the vector.
myVector.resize(5);

// Resizes vec to 8 elements. If size is greater, adds 100s.
myVector.resize(8, 100);

myVector.clear(); // Removes all elements from vec
```

- **Best Practices:**
  - Be aware of iterator invalidation when modifying vector contents.
  - Use **reserve()** before multiple **push_back()** or **emplace_back()** calls to optimize memory reallocations.

```
std::vector<int> myVector2;
myVector2.reserve(10);  // Reserves space for 10 elements
for (int i = 0; i < 10; ++i) {
    myVector2.push_back(i * 10); // No reallocation happens here
}
```

- **Size of a Vector:**
  - The size of a vector refers to the number of elements it currently holds.
  - Accessed using the **size()** method.
- **Capacity of a Vector:**
  - The capacity of a vector is the amount of space allocated for it, which may be equal to or greater than the size.
  - Accessed using the **capacity()** method.
- **Resizing a Vector:**
  - resize**()** changes the size of the vector. If the new size is larger, the vector is extended and new elements are added.
  - reserve**()** increases the capacity of the vector to a specified value if it is greater than the current capacity.
  - This can optimize performance by reducing the number of memory reallocations.
  - shrink_to_fit**()** requests the removal of unused capacity. It's a non-binding request to reduce memory waste.

- **Sorting a Vector:**
  - The std::sort function from the **<algorithm>** header is commonly used.
  - Sorts the elements in a specified range (usually the entire vector).

```
std::vector<int> myVector = {30, 10, 20};
std::sort(myVector.begin(), myVector.end());
```

- **Custom Sort Order:**
  - You can define a custom comparison function for sorting.

```
bool descending(int a, int b) { return a > b; }; // defined elsewhere in the code
std::sort(myVector.begin(), myVector.end(), descending); // Sorts myVector in descending order
```

- **Searching in a Vector:**
  - Use std::find to search for an element.
  - Returns an iterator to the found element or **end()** if not found.

```
std::vector<int>::iterator it = std::find(myVector.begin(), myVector.end(), 20);
if (it != myVector.end()) {
    std::cout << *it << " ";
}
```

Comparison Operators

- **std::vector** supports various comparison operators.
- Enables direct comparison of two vectors.
- Supported Comparison Operators
  - == (Equality)
  - __ (Inequality)
  - __ (Less than)
  - __ (Less than or equal to)
  - __ (Greater than)
  - __ (Greater than or equal to)

```
std::vector<int> myVector1 = {1, 2, 3};
std::vector<int> myVector2 = {1, 2, 3};
std::vector<int> myVector3 = {1, 2, 4};

std::cout << "myVector1 == myVector2: " << (myVector1 == myVector2) << std::endl;
std::cout << "myVector1 != myVector3: " << (myVector1 != myVector3) << std::endl;
std::cout << "myVector3 > myVector1: " << (myVector3 > myVector1) << std::endl;
// Output will be: 1 (true), 1 (true), 1 (true)
```

- How Comparisons Work
  - Element-wise Comparison: Vectors are compared based on their corresponding elements.
  - Order: Comparison starts from the first element and proceeds sequentially.
  - Short Circuiting: Stops at the first unequal pair of elements.
- Key Points
  - **Equality**: Two vectors are equal if they have the same size and all corresponding elements are equal.
  - **Inequality**: The opposite of equality.
  - **Relational Operators**: Compare element by element, similar to comparing words in a dictionary.

```cpp
std::vector<int> myVector4 = {3, 2, 1};
std::vector<int> myVector5 = {1, 2, 4};
std::cout << "myVector4 > myVector5: " << (myVector4 > myVector5) << std::endl;
// Output will be: 1 (true)
```

- Use Cases
    - Useful for sorting algorithms.
    - Comparing state or data contained within vectors.
    - Implementing data structures that rely on ordering (e.g., sets, maps).
- Best Practices
    - Ensure vectors are of comparable types.
    - Be cautious with floating-point comparisons due to precision issues.

- Use Cases
    - Useful for sorting algorithms.
    - Comparing state or data contained within vectors.
    - Implementing data structures that rely on ordering (e.g., sets, maps).
- Best Practices
    - Ensure vectors are of comparable types.
    - Be cautious with floating-point comparisons due to precision issues.

- Dynamically allocating **std::vector** using the **new** keyword.
    - Accessing vector methods via the arrow (**->**) operator.

Vectors in Dynamic Memory

- Why Create a **std::vector** Dynamically?
    - **Flexibility**: Managing the vector's lifetime manually.
    - **Advanced Use-Cases**: Suitable for certain design patterns or complex data structures.
- Creating a **std::vector** Dynamically

```cpp
std::vector<int> *dynamicVector = new std::vector<int>();
```

- Allocates a **std::vector** object on the heap.
- **dynamicVector** is a pointer to the **std::vector** instance.
- Accessing Vector Methods with Arrow Operator
- The arrow operator (->) is used to access members of an object through a pointer.

```cpp
// Adding elements
dynamicVector->push_back(10);
dynamicVector->push_back(20);
// Accessing an element
int firstElement = dynamicVector->at(0);
// Getting the size
size_t vectorSize = dynamicVector->size();
// Iterating over elements
for (auto it = dynamicVector->begin(); it != dynamicVector->end(); ++it) {
    std::cout << *it << " ";
}
// Cleaning up
delete dynamicVector;
```

- Key Points
  - Remember to **delete** the dynamically allocated vector to avoid memory leaks.
  - Accessing methods and members via the arrow operator is essential for pointers to objects.
- Advantages and Considerations
  - **Advantages**: More control over object lifetime and memory management.
  - **Considerations**: Requires careful memory management to avoid leaks.
  - **Best Practices**: Prefer automatic memory management (stack allocation) unless necessary.