C++ `vector` object

Source: https://www.cplusplus.com/reference/vector/vector/

Vectors are:

- Sequence of values like an array
- Dynamically growing or shrinking
- Efficient Random Access. **Use contiguous storage** locations for their elements (like arrays), that means that their elements can be accessed using offsets on pointers to its elements, and just as efficiently as in arrays.

Vector Advantages over arrays:

- Dynamically growing or shrinking

Vector Disadvantages with respect to arrays.

- When size grows/shrinks, all data require copying to new allocate storage and this adds to computational cost. Though this is not done for every element addition or removal request.

Constructors
`vector();` Creates empty vector object with no elements. Example: `vector<double> list1;`

`vector(size_type n, const value_type& val = value_type());`
Creates a vector of specified size and initialized second parameter.
Note that second parameter is optional.
Examples:
`vector<double> list2(10);`
`//size 10, initialized with 0`
`vector<double> list3(5, 100);`
`//size 5, initialized with 100`

`vector(const vector& x);`
Copy Constructor. Creates new vector, copy of parameter vector
Examples:
`vector<double> list4( list1 );`
`vector<double> list5 = list2;`

Accessor Methods
`size_type size();` Returns size of vector. Examples: `cout << list1.size() << endl;`

`bool empty();`
Returns whether the string is empty (i.e., whether its length is 0).
Example: `if ( !list1.empty ) { //do something }`

Mutator Methods
`void push_back(const value_type& val);`

Adds a new element at the end of the vector. The content of val is copied to the new element.

This increases the vector size by one, which causes an automatic reallocation of the allocated storage if required.
Example: `list1.push_back(11.83);`

`void pop_back();`
Removes the last element in the vector, effectively reducing the container size by one.
Example: `list1.pop_back();`

`void clear();`
Removes all elements from the vector (which are destroyed), leaving the container with a size of 0.

`operator=`
Assignment. RHS vector is assigned to left side vector. Size, data everything changes as per the assigned one.
`vector2 = vector1;`

Methods returning references to internal fields (and hence data)

`operator[]`
Examples:
```
cout << list1[3] << list1.at(5) << endl;
vector1[3] = 34.12;
vector1.at(4) = 5.31;
```

`reference front();`
Examples:
```
cout << "First element: " << list1.front() << endl;
```
Note front() returns reference here, therefore following will do what you can guess? `list1.front() = 11.11;`

`reference back()`
Examples:
```
cout << "Last element: " << list1.back() << endl;
```
Note front() returns reference here, therefore following will do what you can guess? `list1.back() = 9.9;`

Iteration

`iterator begin();`
Returns an iterator pointing to the first element in the vector.
`iterator end();`
Returns an iterator referring to the past-the-end element in the vector.
Example:
```
for (vector<double>::iterator it = list1.begin();
        it != list1.end(); ++it)
    cout << ' ' << *it;
```

`iterator rbegin(); //reverse beginning`
`iterator rend(); //reverse end`

`iterator erase(iterator position);`
`iterator erase(iterator first, iterator last);`
Removes from the vector either a single element (position) or a range of

elements ([first, last)).

This reduces the container size by the number of elements removed.

Because vectors use an array as their underlying storage, erasing elements in positions other than the vector end causes the container to relocate all the elements after the segment erased to their new positions. This is generally an inefficient operation compared to the one performed for the same operation by other kinds of sequence containers (such as list or forward_list).

```
iterator insert(iterator position, const value_type& val);
```

The vector is extended by inserting new elements before the element at the specified position, effectively increasing the vector size by 1.
Returns an iterator that points to the newly inserted element.

Examples:

```
list1.insert( list1.begin()+2, 56.99);
```

Relational Operators
`==, !=, <, <=, >, >=` LHS and RHS are vector objects. For **equality**, comparison goes as following: if size is same, actual elements are compared sequentially; stops when mismatch is found. For others comparison is done sequentially and stops wherever it is decisive.