Handout on Vector Class

Vectors are sequence containers that can change in size and are implemented using arrays for fast random access. Like arrays, elements can be accessed sequentially, or directly using offsets on the pointers to its elements. Unlike arrays, the storage can be handled automatically by the container.

Header file

To use this class, we need to include the following header file in our program

#include<vector>

All the member functions of the vector class and the vector class itself are in the namespace std. Vector objects can be constructed based on the contructor used. Various ways in which a vector object can be created, are as follows:

Usage	**Explanation**
vector<int> v1;	This creates an empty vector container of type integer.
vector<int> v2 (10,3);	This creates a vector container of 10 integers with all set to value 3.
vector<int> v3 (v2.begin(),v2.end());	This creates a vector container **v3** of 10 integers with all set to value 3 by iterating though the vector container **v2** created in the previous example.
vector<int> v4 (v3);	This copies the vector container **v3** to v4.

The iterator constructor can also be used to construct vector containers from arrays.

For example, we can initially define an array as **int A[]={3, 6, 4, 7};** and use the iterator construct as **vector<int> v5 (A, A + sizeof(A)/sizeof(int));**

The member function **operator=** can be used to assign new contents to the vector container. The following statements will initialize the size of vector container **b2** as 4(b1 initially declared as size 4 will all 4 integers initialized to 0) with all 4 integers initialized to 0.

vector<int> b1 (4,0);

vector<int> b2;

b2 = b1;

Iterators:

Iterators are public member functions defined for iterating through the element of type defined for the containers. Following are the list of iterators available for the vector class.

begin()

This returns an iterator pointing to the first element in the vector. The following statement will initialize the iterator pointing to the begining of the vector container

vector<int> b1(4,0);

vector<int>::iterator it = b1.begin();

We can use use ***it** to access the first element of the vector container **b1**.

end()

This returns an iterator pointing to the past-the-end element in the vector. The following statement will initialize the iterator pointing to the past-the-end element of the vector container

vector<int> b1(4,0);

vector<int>::iterator it = b1.end();

We cannot use use ***it** to access, as it will not point to any element of the vector container, and should not be used to dereference.

begin in conjunction with end can be used to iterate through the vector elements, as follows:

```
for (vector<int>::iterator it = v5.begin(); it != v5.end(); ++it) {
   cout << ' ' << *it;
}
```

rbegin()

rbegin is a backward iterator, and it returns a reverse iterator pointing to the last element in the vector. It points to the element before the one that would be pointed to by member iterator end and incrementing it moves towards the begining of the container. The following statement will initialize the reverse iterator pointing to the last element of the vector container

vector<int> b1(4,0);

vector<int>::reverse_ iterator rit = b1.rbegin();

After initializing, we can use ***rit** to access the last element of the vector container.

rend()

rend is a backward iterator and it returns a reverse iterator pointing to the element before the first element in the vector. The following statement will initialize the reverse iterator pointing to the element one before the first element

vector<int> b1(4,0);

vector<int>::reverse_ iterator rit = b1.rend();

After returning a reverse iterator, we should not use ***rit** to access the element of the vector container, as it will point to an element before the first element of the vector container.

rbegin in conjunction with rend can be used to reverse iterate through the vector elements, as follows:

```
for (vector<int>::reverse_iterator rit = v5.rbegin(); it != v5.rend(); ++rit) {
   cout << ' ' << *rit;
}
```

Capacity:

When vectors are initialized, they typically consist of a pointer to a dynamically allocated memory. The allocated size(capacity) may be larger than the actual size used in the program. When new elements are inserted, the actual size of the vector is automatically set. If the size becomes larger than the capacity, reallocation occurs. We mention below some of the useful member functions for checking size and capacity of the vector container.

size()

size is a member function which returns the actual size(number of elements) of the vector in use by the program. The following statements can be used to calculate the actual size of the vector container. In this example, the size returned is 3.

```
vector<int> b1(3,4);
 cout << b1.size();
```

If we had used **vector<int> b1;** instead of the above statement, the size returned would have been 0.

capacity()

This returns the size of the allocated space for the vector during initialization. It can be equal or greater than the actual size. For example, if the statements are

```
vector<int> b1(50,4);
cout<< b1.capacity();
```

Then, the cout statement could possibly return 128.

max_size()

This returns the maximum number of elements the vector can hold. It is system dependent. On some systems, you would get the value as high as 1073741823.

empty()

This can be used to test if the vector is empty(no elements). The function returns a boolean result.

Usage:	**Explanation**
if(!v3.empty()){ **//do this** **}** **else{** **//do something** **}**	**v3** is a vector container checked to see whether it is not empty

resize()

This member function resizes the vector container so that it contains the number of elements(n) specified in the argument. The argument can also contain the value that is to be written for each of the **n** elements. If new **n** is greater than the current size of the container, then the container is expanded by adding those many elements(value if specified in the argument) required to make the new size of the container equal to **n**. If **n** is less than the previous size of the container, then the container is resized to **n** with the remaining elements destroyed.

Usage	**Explanation**
vector<int> b1(10,3);	create a vector container of 10 integers with all set to value 3.
b1.resize(5);	resizes the size of container to 5 and destroys all the elements
b1.resize(7,20);	resizes the size of container to 7 by adding two elements(both set to value 20)

Element Access:

Elements of a vector container, can be accessed using various mechanisms as described below:

operator[]

It returns a reference to the position of an element in the container. The position starts with 0(like in basic C arrays) rather than 1. The operator[] allows the vector container to be accessed in the

same fashion as the array element access using the index. The member operator[] does not check for bounds, and has undefined behavior if access is made using a position value which is out of bound.

The following statements of code will illustrate the usage of an **operator[]**

Usage	**Explanation**
vector<int> v5(10,3);	This creates a vector container **v5** of 10 integers with all set to value 3.
for(int k = 0 ; k < v5.size(); k++{ **v5[k]=4;** **}**	The for loop assigns value 4 to all the elements of **v5**

at( )

This is a member function taking argument as position value **n**, and has the same effect on accessing vector elements using **operator[]**, except that out_of_range exception is thrown if access is made to a non-valid(out of bound) vector element

The following statements of code will illustrate the usage of an **at( )** member function

Usage	**Explanation**
vector<int> v5(10,3);	This creates a vector container **v5** of 10 integers with all set to value 3.
for(int k = 0 ; k < v5.size(); k++{ **v5.at(k)=4;** **}**	The for loop assigns value 4 to all the elements of **v5**

front( )

This member function returns a direct reference to the first element in the vector, unlike begin() which returns an iterator(abstract pointer). It has an undefined behavior if you try to call this function on an empty container.

The following example of code will illustrate the usage of a **front( )** member function

Usage	**Explanation**
vector<int> v5(10,3);	This creates a vector container v5 of 10 integers with all set to value 3.

for(int k = 0 ; k < v5.size(); k++{
v5.at(k)=k;
}
	The for loop assigns value of counter k to each element of v5
	i.e., v5[0]=0, v5[1]=1, v5[2]=2, v5[3]=3, v5[4]=4

cout<<v5.front();	This will display 0 as it was assigned to the 0th element of the vector v5

back( )

This member function returns a direct reference to the end element in the vector, unlike **end()** which returns an iterator(abstract pointer) **past-the-end** element. It has an undefined behavior if you try to call this function on an empty container.

The following example of code will illustrate the usage of a **back( )** member function

Usage	**Explanation**
vector<int> v5(10,3);	This creates a vector container **v5** of 10 integers with all set to value 3.
for(int k = 0 ; k < v5.size(); k++{	The for loop assigns value of counter **k** to each element of **v5**

v5.at(k)=k;	i.e., **v5[0]=0, v5[1]=1, v5[2]=2, v5[3]=3, v5[4]=4**
}	
cout<<v5.back();	This will display 4 as it was assigned to the 4th element of the vector **v5**

Modifiers:

These are member functions which help in assigning values to elements of vector containers, add an element at the end of a vector, delete the last element, insert elements at arbitrary position, erase elements, etc. The following member functions are used to perform such operations:

assign( )

This member function assigns new contents to the vector by replacing its current content, and modifying its size accordingly.

The following example code will illustrate the usage of an **assign( )** member function with varous parameters:

Usage	**Explanation**
vector<int> v1, v2, v3;	This creates 3 empty vectors **v1**, **v2**, **v3** of type integer
v1.assign(8,10);	This assigns 8 integer elements with a value of 10 each.
cout<<v1.size();	This will display 8 as 8 integer values were assigned to **v1**
vector<int>::iterator it;	This declares an iterator it to a vector of type integer
it= v1.begin()+2;	iterator it pointing to the third element of **v1**
v2.assign(it, v1.end()-2);	This will assign four elements from **v1** to **v2**, starting from the third element of **v1** to the sixth element of **v1** Note: **v1.end()** returns an iterator pointing to the past-the-end element in the vector **v1**.
cout<<v2.size();	This will display 4 as 4 integer values were assigned from **v1**
int A[]={1,2,3};	Array **A** is declared and initialized to 3 elements
v3.assign(A,A+3);	vector **v3** gets assigned from array **A**
cout<<v3.size();	This will display 3 as 3 integer values were assigned from array **A**

push_back( )

This memeber function accepts a single value(this value depends on the type, i.e., int, float), and adds a new element at the end to the vector.

The below example illustrates the push_back function:

Usage	**Explanation**
vector<int> v5(2,3);	This creates a vector container **v5** of 2 integers with all set to value 3.
for(int k = 0 ; k < v5.size(); k++{	The **push_back** function adds each value of counter **k** to the
v5.push_back(k);	end of **v5** i.e., **v5[0]=3, v5[1]=3, v5[2]=0, v5[3]=1**
}	and note that **v5.size()** will now be 4.

pop_back( )

This deletes the end element of the vector, thus reducing the size by 1. In the above example, if **v5.pop_back();** is executed, the size of vector **v5** will be reduced by 1 i.e., 3 and the elements of the vector **v5** will be **v5[0]=3, v5[1]=3, v5[2]=0**

insert( )

The vector container can also be extended by inserting new elements based on the position(position between the first element and the last element) specified. The elements are inserted before the element at the specified position. The size of the vector is automatically increased. This operation is very inefficient as all the elements after the position specified need to be reallocated to make space for the new elements.

The below example illustrates the **insert()** function in various ways:

Usage	**Explanation**
vector<int> v5(2,3);	This creates a vector container **v5** of 2 integers with both set to value 3. This makes **v5[0]=3, v5[1]=3**
vector<int>::iterator it =v5.begin();	initializing iterator it
v5.insert(it, 2, 4);	inserting 2 elements of value 4 before the first element(pointed by iterator it at the first element) of the vector **v5**. This makes **v5[0]=4, v5[1]=4, v5[2]=3, v5[3]=3**
it = v5.begin();	iterator **it** needs to be re-initialized as the old one is not valid
int A[] ={2,2};	Array A declared and initialized with two elements each of value 2
v5.insert(it+2, A, A+2);	The iterator **it** is increased by 2 and **it** will point to **v5[2]**. This will insert the two elements of the array A, before the element at **v5[2]**. This makes **v5[0]=4, v5[1]=4, v5[2]=2, v5[3]=2, v5[4]=3, v5[5]=3.**

erase( )

The element(s) from the vector container, can also be removed by using this function. It can either remove a single element or a range of elements specified in the arguments. In using the range from first and last as an argument, it will remove all the elements positioned between the first and the last, including the element pointed by first(but not last). The size of the vector container will be automatically decreased.

The below examples illustrate the **erase()** function:

Usage	**Explanation**
vector<int> v5(4,3);	This creates a vector container **v5** of 4 integers all set to value 3. This makes **v5[0]=3, v5[1]=3, v5[2]=3, v5[3]=3**
vector<int>::iterator it =v5.begin();	initializing iterator it
v5.insert(it+2, 1, 4);	The iterator it is increased by 2 and it will point to **v5[2]**. This will insert an element having value 4 before the element at **v5[2]** This makes **v5[0]=3, v5[1]=3, v5[2]=4, v5[3]=3, v5[4]=3**
v5.erase(it+2);	This will erase the 3rd element, the size of vector **v5** will be 4. This makes **v5[0]=3, v5[1]=3, v5[2]=3, v5[3]=3**

swap( )

This member function exchanges the content of the container by the content of another container of the same type specified in the argument. The iterators reamain valid even after swapping vector contents.

Usage	**Explanation**
vector<int> v5(4,3);	This creates a vector container **v5** of 4 integers all set to value 3. This makes **v5[0]=3, v5[1]=3, v5[2]=3, v5[3]=3** The size of **v5** is 5
vector<int> v6(2,5);	This creates a vector container **v5** of 2 integers both set to value 5. This makes v**6[0]=5, v6[1]=5** The size of **v6** is 2
v5.swap(v6);	This will make **v6** as size 4 containing 4 elements each of value 3 and **v5** as size 2 containing 2 elements both having value 5

For more details, please refer to the following reference links:

http://www.cplusplus.com/reference

http://en.wikipedia.org/wiki/C++_Standard_Library