Handout on List Class

The template library 'list' is a sequence container that contains elements of different types. It is implemented like a double linked list, where different elements are stored at different locations, but they are linked with each other and the order depends on their linking. Every element in the list will have a link to the previous element and the next element. If it is the first element, then there is no 'previous link' and if it is last one, then there is no 'next link'. Objects of the list are known as container objects. This container allows us to traverse, insert, delete, the elements of the list.

Header file

To use this class, we need to include this as a header file in our program

```
#include<list>
```

Creating List Objects / Variables

List objects are created as:

```
list <datatype> <list_object_name>...;
list <int> marks(10,0)
```

In this example, we have defined a list named 'marks' with '10' elements, where all the 10 elements of the list 'marks' have been assigned value '0'.

Given below is a list of functions of this class. Assume that all the header files are included and the objects(used in the table below) of string class have been already created. **Henceforth, the objects of the list will be represented as 'lob' while describing the syntax.**

Empty Constructor

list<*datatype*> lob;

This is an empty constructor that creates an empty list named 'lob'.

E.g.

```
   list<int> item;
```

Constructors

1 Empty Constructor

list<*datatype*> lob;

This is an empty constructor that creates an empty list named 'lob' of the datatype mentioned in angular brackets.

E.g.

```
   list<int> item;
```

This creates a list 'item' of type integer.

2 Fill Constructor

list<*datatype*> lob(int n_elements, int value);

This creates a list 'lob' of 'n_elements' and initializes those 'n_elements' of the list with 'value'.

E.g.

```
   list<int>item(100,-4)
```

This creates a list 'item' having 100 integers, all initialized to value -4.

3 Copy Constructor

list<int> lob2(lob1);

This constructor creates a copy of lob1 and stores in lob2 in the same order given in 'lob1'.

E.g.

```
   list<int> item(100,-4)
   list<int> item_copy(item);
```

The first line creates a list 'item'. The second line creates a list 'item_copy' and copies the list 'item' to the list 'item_copy'.

6 Range Constructor

list<int> lob (Iterator range 1, Iterator range 2);

This constructor creates a list 'lob' and copies the elements from range1 to range2 by maintaining the same order in which they were present. 'Iterator' is an object.

E.g.

```
list<int> item(100,-4)
list<int> item_copy(item.begin(), item.end());
```

The first statement, creates a list 'item'. The second one creates a list 'item_copy' and copies the elements of list 'item' from beginning to the end. 'begin()' and 'end()' are functions that points to the beginning of the list and end of the list respectively. These functions and the concept of iterators will be covered next.

The values from the arrays can also be copied to the list.

E.g.

```
int array1[] = {10, 20, 30, 40, 50};
list<int> item(array1, array1 + 3);
```

In this example, the integer array 'array1' contains 5 elements. 'array1' points to the first element in the array i.e. '10' and 'array1 + 3' points to the third element in the array 'array1' i.e. 30. After execution of the second statement, the list 'item' will contain 3 elements in the same order of 'array1', i.e. '10, 20, 30'

Iterators

Iterator is an object which points to an element in the list i.e. a container. It allows the programmers to traverse through the list i.e. a container using different functions. The list of iterator functions are given below.

7 begin

list<*datatype*>::iterator *itname* = lob.begin();

The iterator 'begin' returns the iterator that points to the first element in the list.

E.g.

```
int array1[] = {10,20,30,40,50};
list<int> item(array1, array1+3);
list<int>::iterator it = item.begin();
cout << "1st item = " << *it <<endl;
```

Output

```
1st item = 10
```

8 end

list<*datatype*>::iterator *itname* = lob.end();

The iterator 'end' returns the iterator that points to the next element after the last element in the list. For example if my list is 10, 20, 30, this iterator will point to the location after the last element '30' in the list. To access the last element, you must first use this iterator and then decrement the iterator by 1.

E.g.

```
int array1[] = {10,20,30,40,50};
list<int> item(array1, array1+3);
list<int>::iterator it = item.end();
it--;
cout << "Last item = " << *it <<endl;
```

Output

```
Last item = 30
```

9 rbegin

list<*datatype*>::reverse_iterator *itname* = lob.rbegin();

'rbegin' returns a 'reverse_iterator' that points to the last element in the string. This iterator increments backwards. For example, if the last element is at position 10, 'reverse_interator' will return 10. If this was incremented by 1, instead of pointing at 11 position, it will point to 9. This will be incremented till the reverse_iterator points to beginning of the list.

E.g.
```
   int array1[] = {10,20,30,40,50};
   list<int> item(array1, array1+3);
   list<int>::reverse_iterator it = item.rbegin();
   cout << "Item = " << *it <<endl;
   it++;
   cout << "Item = " << *it <<endl;
   it++;
   cout << "Item = " << *it <<endl;
```
Output
```
  Item = 30
  Item = 20
  Item = 10
```

10 rend

list<*datatype*>::`reverse_iterator` *itname* = lob.rend();

'rend' will return a 'reverse_iterator' that points to the previous element of the first element in the list. To access all the elements of the list from the first element till the end, we need to decrement the 'reverse_iterator'. For example, 'rend' will initially point to the element preceeding the first element. If we want to access the first element, we will decrement the 'reverse_iterator' by 1. Now, since the 'reverse_iterator' is pointing to first element, if we again decrement by 1, we will now point to the second element in the list.

E.g.
```
   int array1[] = {10,20,30,40,50};
   list<int> item(array1, array1+3);
   list<int>::reverse_iterator it = item.rend();
   it--;
   cout << "Item = " << *it <<endl;
   it--;
   cout << "Item = " << *it <<endl;
   it--;
   cout << "Item = " << *it <<endl;
```
Output
```
  Item = 10
  Item = 20
  Item = 30
```

Capacity

11 Size

size_t lob.size();

This returns the number of elements in the list

E.g.
```
   int array1[] = {10,20,30,40,50};
   list<int> item(array1, array1+3);
   cout << "Number of elements in the list = " << item.size() <<endl;
```
Output
```
  Number of elements in the list = 3
```

12 empty

bool lob.empty()

This returns whether the list is empty or not. A list is called as empty when it has 0 element. It returns true if it is empty, else it returns false.

E.g.

```
int array1[] = {10,20,30,40,50};
list<int> item1(array1, array1+3);
list<int> item2;
cout << "Is item1 list empty? " << item1.empty() << endl;
cout << "Is item2 list empty? " << item2.empty() << endl;
```

Output

```
Is item1 list empty? 0
Is item2 list empty? 1
```

Traversing all the elements in the List (From 1st to last element)

E.g.

```
int array1[] = {10,20,30,40,50};
list<int> item(array1, array1+3);
int count=0;

cout << "Items in the list:" << endl;
for (list<int>::iterator it=item.begin(); it != item.end(); ++it) {
   count++;
   cout << "Item "<< count << ": " << *it << endl;
}
```

Output

```
Items in the list:
Item 1: 10
Item 2: 20
Item 3: 30
```

Traversing all the elements in the List (From last element to the first element)

E.g.

```
int array1[] = {10,20,30,40,50};
 list<int> item(array1, array1+3);
 int count=0;
 cout << "Items in the list:" << endl;
 for(list<int>::reverse_iterator it=item.rbegin();it!=item.rend();++it)
 {
    count++;
    cout << "Item "<< count << ": " << *it << endl;
 }
```

Output

```
Items in the list:
Item 1: 30
Item 2: 20
Item 3: 10
```

Element Access
Using the functions of 'Element Access', we can access the elements of the list.

13 back
lob.back()
This returns the reference to the last element in the list.
E.g.
```
int array1[] = {10,20,30,40,50};
list<int> item(array1, array1+3);
cout << "Last item: " << item.back() << endl;
```
Output
```
Last item: 30
```

14 front
lob.front()
This returns the reference to the first element in the list.
E.g.
```
int array1[] = {10,20,30,40,50};
list<int> item(array1, array1+3);
cout << "Last item: " << item.back() << endl;
cout << "First item: " << item.front() <<endl;
```
Output
```
Last item: 30
First item: 10
```

Modifiers

15 **Push back**
a) lob.push_back(value)
b) lob.push_back(variable_name)
Both, (a), and (b), adds a new element mentioned in parenthesis at the end of the list, i.e. after the last element in the list. The size of the variable increases to 1 after executing this function.
E.g.
```
int array1[] = {10,20,30,40,50};
list<int> item(array1, array1+3);
int num = 999;
item.push_back(888);
item.push_back(num);
cout << "Items in the list:" << endl;
for (list<int>::iterator it=item.begin(); it != item.end(); ++it) {
   cout << *it << " " ;
}
```
Output
```
Items in the list:
10 20 30 888 999
```
In this example, at first the list contains the three elements from the array 'array1', i.e. 10, 20, 30. Now, 'push_back(888)' is executed. This will insert '888' at the end of the list and the size of the list is increased by 1. So, the modified list is 10, 20, 30, 888. Since, variable 'num' has value 999, and when push_back(num) executes, 999 is inserted after 888 and the size of the list is increased by 1. The final list is then printed.

16 **Push front**
a) lob.push_front(value)
b) lob.push_front(variable_name)
Both, (a), and (b), adds a new element mentioned in parentheses at the beginning of the list, i.e. before the first element in the list. The size of the variable increases to 1 after executing this function.
E.g.

```
int array1[] = {10,20,30,40,50};
list<int> item(array1, array1+3);
int num = -333;
item.push_front(-5);
item.push_front(num);
cout << "Items in the list:" << endl;
for (list<int>::iterator it=item.begin(); it != item.end(); ++it) {
   cout << *it << " " ;
}
```

Output

```
Items in the list:
-333 -5 10 20 30
```

In this example, at first the list contains the three elements from the array 'array1', i.e. 10, 20, 30. Now, 'push_front(-5)' is executed. This will insert '-5' at the beginning of the list and the size of the list will be increased by 1. So, the modified list is -5, 10, 20, 30. Since, variable 'num' has value -333, and when push_front(num) executes, -333 is inserted before -5 and the size of the list is increased by 1. The final list is then printed.

17 **Pop back**
lob.pop_back();
This removes the last element from the list and decreases the size of the list by 1
E.g.

```
int array1[] = {10,20,30,40,50,60,70,80,90};
list<int> item(array1, array1+8);

cout << "Before removing the last item:" << endl;
for (list<int>::iterator it=item.begin(); it != item.end(); ++it) {
   cout << *it << " " ;
}
item.pop_back();
item.pop_back();
cout << endl << "After removing the last two items:" << endl;
for (list<int>::iterator it=item.begin(); it != item.end(); ++it) {
   cout << *it << " " ;
}
```

Output

```
Before removing the last item:
10 20 30 40 50 60 70 80
After removing the last two items:
10 20 30 40 50 60
```

In this example, the list will contain '10,20,30,40,50,60,70,80'. When 'item.pop_back' is executed, it removes the last element i.e. '80' from the list and decreases the size of the list by 1. So, the updated list is '10,20,30,40,50,60,70'. Once, again, pop_back() is executed. This will now remove the element 70 from the list and will decrease the size of the list by 1. This updated list is then printed.

18 **Pop front**
lob.pop_front();
This removes the first element from the list and decreases the size of the list by 1
E.g.
```
int array1[] = {10,20,30,40,50,60,70,80,90};
list<int> item(array1, array1+8);

cout << "Before removing the last item:" << endl;
for (list<int>::iterator it=item.begin(); it != item.end(); ++it) {
   cout << *it << " " ;
}
item.pop_front();
item.pop_front();
cout << endl << "After removing the last two items:" << endl;
for (list<int>::iterator it=item.begin(); it != item.end(); ++it) {
   cout << *it << " " ;
}
```
Output
```
Before removing the last item:
10 20 30 40 50 60 70 80
After removing the last two items:
30 40 50 60 70 80
```
In this example, the list will contain '10,20,30,40,50,60,70,80'. When 'item.pop_front' is executed, it removes the first element i.e. '10' from the list and decreases the size of the list by 1. So, the updated list is '20,30,40,50,60,70'. Once, again, pop_front() is executed. This will now remove the element 20 from the list and will decrease the size of the list by 1. This updated list is then printed.

19 **Clear**
lob.clear();
Unlike the function 'pop_back' and 'pop_front' that removes one element from the list, 'clear' removes all the elements from the list. Thus, after executing this function, the list would be an empty list containing 0 elements.
E.g.
```
int array1[] = {10,20,30,40,50,60,70,80,90};
list<int> item(array1, array1+8);
cout << "Size of list: " << item.size() << endl;
item.clear();
cout << "Size of list: " << item.size() << endl;
```
Output
```
Size of list: 8
Size of list: 0
```

20 **Insert**
a) lob.insert(iterator, value);
b) lob.insert(iterator, nelements, value);
(a) This inserts the the new element mentioned in 'value' before the current element
(b) This inserts 'nelements' new elements in the list with the value mentioned before the current element.
E.g.
```
int array1[] = {10,20,30,40,50,60,70,80,90};
list<int> item(array1, array1+8);
list<int>::iterator it = item.begin();
it++;   //1st increment
item.insert(it,555);
it++;   //2nd increment
item.insert(it,3,-66);
```

```
    cout << "Elements inserted in the list: " << endl;
    for (it=item.begin(); it != item.end(); ++it) {
       cout << *it << " " ;
    }
```
Output

```
Elements inserted in the list:
10 555 20 -66 -66 -66 30 40 50 60 70 80
```

In this example, at first the list contains '10,20,30,40,50,60,70,80,90'. The iterator points to the first element i.e. '10' in the list. Now, the iterator value is increased by 1 (see 1st increment comment), i.e. it now points to 2nd element i.e. 20. Now, 'item.insert(555)' is executed, which will insert the number '555' before the current element '20'. So the list becomes '10, 555, 20, 30, 40, 50, 60, 70, 80, 90'. Note, that it is still pointing to the element '20' even though an element was inserted. Now, again the iterator is incremented (See //2nd increment comment). This iterator will now point to 4th element i.e. '30'. Now, 'insert(it,3,-66);' will insert -66, 3 times before the value 30 in the list. So, the updated list is '10 555 20 -66 -66 -66 30 40 50 60 70 80', which is then printed

21 **Erase**

a) lob.erase(iterator)

This removes one element from the list at the position pointed by the iterator.

E.g No. 1.

```
    int array1[] = {10,20,30,40,50,60,70,80,90};
    list<int> item(array1, array1+8);
    list<int>::iterator it1;
    cout << "Original List: " << endl;
    for (it1=item.begin(); it1 != item.end(); ++it1) {
       cout << *it1 << " " ;
    }
    it1 = item.begin();   //This points to the 1st element in the list
    it1++;                //This points to the 2nd element in the list
    it1 = item.erase(it1);  //This erases the 2nd element i.e. 20
    cout << endl << "Erased 2nd element: " << endl;
    for (it1=item.begin(); it1 != item.end(); ++it1) {
       cout << *it1 << " " ;
    }
```
Output

```
Original List:
10 20 30 40 50 60 70 80
Erased 2nd element:
10 30 40 50 60 70 80
```

b) lob.erase(iterator1, iterator2)

This removes the elements of the list starting from the element pointing by iterator1 till the previous element pointing by iterator2. For example, if iterator1 points to 2nd element and iterator2 points to 7th element, then the elements 2, 3, 4, 5, and 6 will be deleted. 7th element will not be deleted.

E.g. No. 2

```
    int array1[] = {10,20,30,40,50,60,70,80,90};
    list<int> item(array1, array1+8);
    list<int>::iterator it1;
    list<int>::iterator it2;
    cout << "Original List: " << endl;
    for (it1=item.begin(); it1 != item.end(); ++it1) {
       cout << *it1 << " " ;
    }
    it1 = item.begin();  //This points to the 1st element in the list
    it1++;               //This points to the 2nd element in the list
```

```
   it2 = item.end();
   it2--;              //This points to last element in the list
   it2--;              //This points to the 2nd last element
   it1 = item.erase(it1,it2);   //Will remove elements 2 to it2-1
   cout << endl << "Erased elements from 2nd to 2nd last: " << endl;
   for (it1=item.begin(); it1 != item.end(); ++it1) {
      cout << *it1 << " " ;
   }
```

Output

```
Original List:
10 20 30 40 50 60 70 80
Erased elements from 2nd to 2nd last:
10 70 80
```

Operations

22 **remove**

lob.remove(value/variable)

Removes all the occurrences of the value mentioned, from the list.

E.g.

```
   int array1[] = {10,20,30,10,50,10,20,80,90};
   list<int> item(array1, array1+8);
   list<int>::iterator it;
   cout << "Original List: " << endl;
   for (it=item.begin(); it != item.end(); ++it) {
      cout << *it << " " ;
   }
   // Removes all the occurrences of value 10 from the list
   item.remove(10);
   cout << endl << "List after deleting value 10: " << endl;
   for (it=item.begin(); it != item.end(); ++it) {
      cout << *it << " " ;
   }
```

Output

```
Original List:
10 20 30 10 50 10 20 80
List after deleting value 10:
20 30 50 20 80
```

23 **reverse**

lob.reverse();

This reverses the order of the elements in the list.

E.g.

```
   int array1[] = {10,20,30,10,50,10,20,80,90};
   list<int> item(array1, array1+8);
   list<int>::iterator it;
   cout << "Original List: " << endl;
   for (it=item.begin(); it != item.end(); ++it) {
      cout << *it << " " ;
   }
   item.reverse();  //Reverses the elements in the list
   cout << endl << "List after reversing: " << endl;
   for (it=item.begin(); it != item.end(); ++it) {
      cout << *it << " " ;
   }
```

Output

```
Original List:
10 20 30 10 50 10 20 80
List after reversing:
80 20 10 50 10 30 20 10
```

24 **sort**

lob.sort();

This sorts the elements of the list.

E.g.

```
int array1[] = {10,20,30,10,50,10,20,80,90};
list<int> item(array1, array1+8);
list<int>::iterator it;
cout << "Original List: " << endl;
for (it=item.begin(); it != item.end(); ++it) {
   cout << *it << " " ;
}
item.sort();
cout << endl << "List after sorting: " << endl;
for (it=item.begin(); it != item.end(); ++it) {
   cout << *it << " " ;
}
```

Output

```
Original List:
10 20 30 10 50 10 20 80
List after sorting:
10 10 10 20 20 30 50 80
```

25 **merge**

lob1.merge(lob2)

To perform merge operation, both the lists should be already sorted. This function will perform merge sorting on both the list and the final list will be updated in 'lob1'.

E.g.

```
int array1[] = {10,40,30,20};
int array2[] = {99, 25, 2, 99};
list<int> item1(array1, array1+4);
list<int> item2(array2, array2+4);
list<int>::iterator it;
item1.sort();
item2.sort();
item1.merge(item2);

cout << "Merged List: " << endl;
for (it=item1.begin(); it != item1.end(); ++it) {
   cout << *it << " " ;
}
```

Output

```
Merged List:
2 10 20 25 30 40 99 99
```

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

http://www.cplusplus.com/reference

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