Maciej Sobieraj

Lecture 11

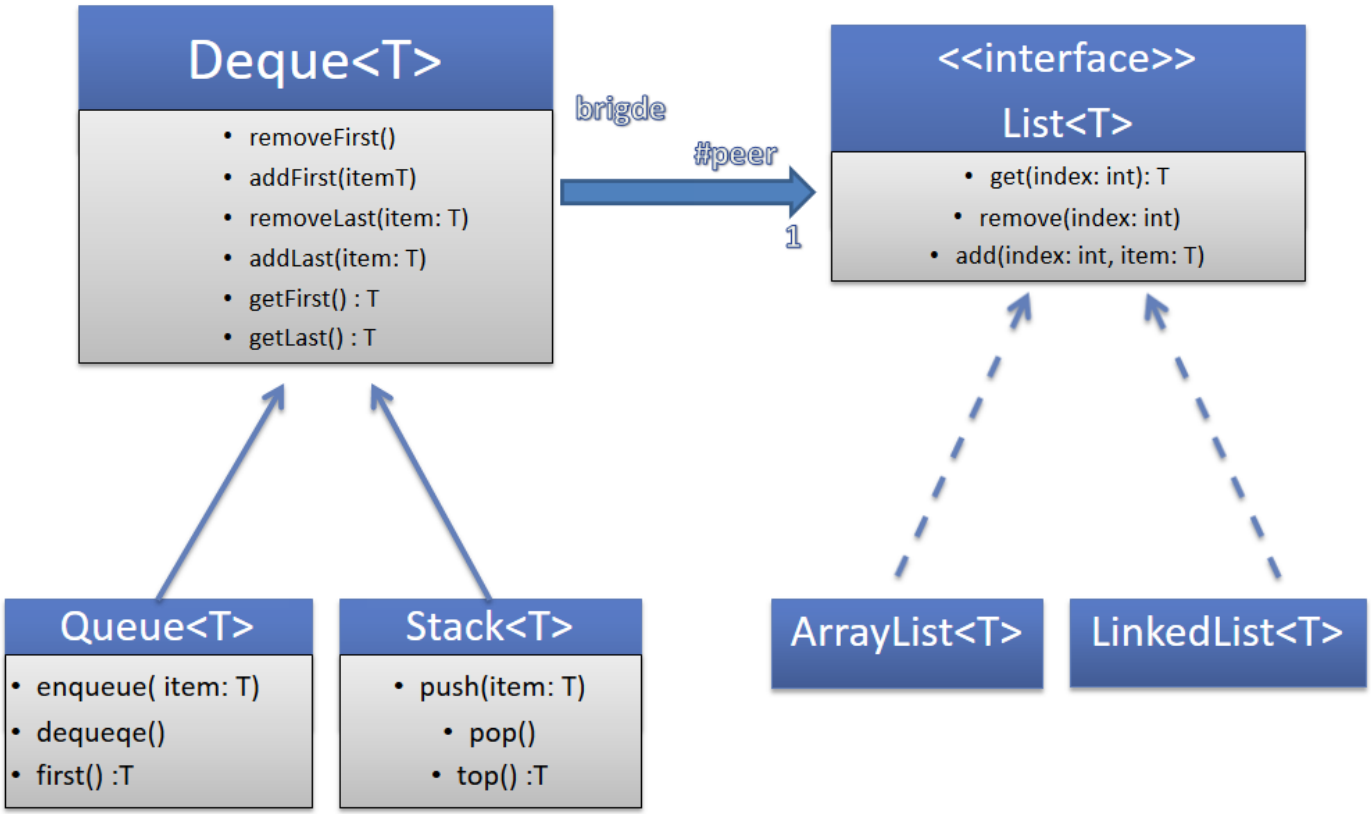
Outline

1. STL sequential containers
    1. **Sequence containers**
    2. Iterators
    3. Operations

deque class

- **Header:**
  - <deque>
- **Definition:**
  - **template<**
  - **class** T,
  - **class** Allocator = std::allocator<T>
  - **> class** deque;
- The name deque stands for **double-ended queue**.

deque class


Deque<T>
removeFirst()
addFirst(itemT)
removeLast(item: T)
addLast(item: T)
getFirst() : T
getLast() : T
brigde
#peer
1
<<interface>>
List<T>
get(index: int): T
remove(index: int)
add(index: int, item: T)
Queue<T>
enqueue(item: T)
dequeqe()
first() :T
Stack<T>
push(item: T)
pop()
top() :T
ArrayList<T>
LinkedList<T>

deque constructors – size constructors

- deque has four constructors, virtually identical to those of the vector:
  - **explicit** deque (**const** Allocator& = Allocator());
  - **explicit** deque
    (size_type n, const T& value= T(), const Allocator& = Allocator());
  - **template** <**class** InputIterator>
    deque (InputIterator first, InputIterator last, const Allocator& = Allocator());
  - deque (**const** deque<T,Allocator>& x);

deque constructors – size constructors

```
#include <deque>
#include <iostream>

using namespace std;

int main()
{
    deque<int> d1(10, 0);
    cout<<"Size: "<<d1.size()<<endl;
    for(unsigned i = 0; i < d1.size(); ++i)
    {
        cout<< d1[i]<<" ";
    }
    cout<<endl;
    return 0;
}
```

```
Output:
Size: 10
0 0 0 0 0 0 0 0 0 0
```

deque – iterator constructors

```
#include <deque>
#include <iostream>

using namespace std;

int main()
{
    int a1[]={1,2,3,4,5,6,7,8,9,10};
    //first one
    deque <int>d1(a1, a1+10);
    cout<<"Size (d1):  "<<d1.size()<<endl;
    for(unsigned i = 0; i < d1.size(); ++i)
    {
        cout<< d1[i]<<" ";
    }
    cout<<endl;
    //second one;
    deque <int>d2(a1+5,a1+10);
    cout<<"Size (d2):  "<<d2.size()<<endl;
    for(unsigned i = 0; i < d2.size(); ++i)
    {
        cout<< d2[i]<<" ";
    }
    cout<<endl;
    return 0;
}
```

```
Output:
Size (d1):  10
1 2 3 4 5 6 7 8 9 10
Size (d2):  5
6 7 8 9 10
```

deque – iterator constructors

```
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

int main()
{
    //vector
    vector <int>v(10, 0);
    for(unsigned i = 0; i < v.size(); ++i)
    {
        v[i]=i+1;
    }
    cout<<"Size (v):  "<<v.size()<<endl;
    for(unsigned i = 0; i < v.size(); ++i)
    {
        cout<< v[i]<<" ";
    }
    cout<<endl;
    //deque
    deque <int>d(v.begin(), v.begin()+5);
    cout<<"Size (d):  "<<d.size()<<endl;
    for(unsigned i = 0; i < d.size(); ++i)
    {
        cout<< d[i]<<" ";
    }
    cout<<endl;
    return 0;
}
```

```
Output:
Size (v):  10
1 2 3 4 5 6 7 8 9 10
Size (d):  5
1 2 3 4 5
```

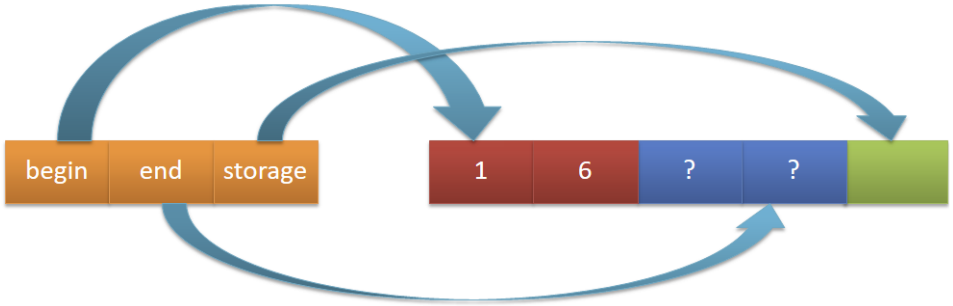
deque – copy constructors

```
#include <deque>
#include <iostream>

using namespace std;

int main()
{
    int a1[]={1,2,3,4,5,6,7,8,9,10};
    //first one
    deque <int> d1(a1, a1+10);
    cout<<"Size (d1):  "<<d1.size()<<endl;
    for(unsigned i = 0; i < d1.size(); ++i)
    {
        cout<< d1[i]<<" ";
    }
    cout<<endl;
    //second one;
    deque <int> d2(d1);
    cout<<"Size (d2):  "<<d2.size()<<endl;
    for(unsigned i = 0; i < d2.size(); ++i)
    {
        cout<< d2[i]<<" ";
    }
    cout<<endl;
    return 0;
}
```

```
Output:
Size (d1):  10
1 2 3 4 5 6 7 8 9 10
Size (d2):  10
1 2 3 4 5 6 7 8 9 10
```

list

- **Header:**
  - <list>
- **Definition:**
  - **template**<
  - **class** T,
  - **class** Allocator = std::allocator<T> >
  - **class** list;



list

- The *list* container is an implementation of the double-linked list principle.
- Each element has pointers leading to the next and the previous ones in a list sequence.
- The *list* container allows for fast insertion and deletion anywhere inside the range of its elements.

Outline

1. STL sequential containers
    1. Sequence containers
    2. **Iterators**
    3. Operations

Iterators

- The iterator is in many ways similar to the concept of the pointer, and in some cases, they can be used interchangeably.

Containers and iterators

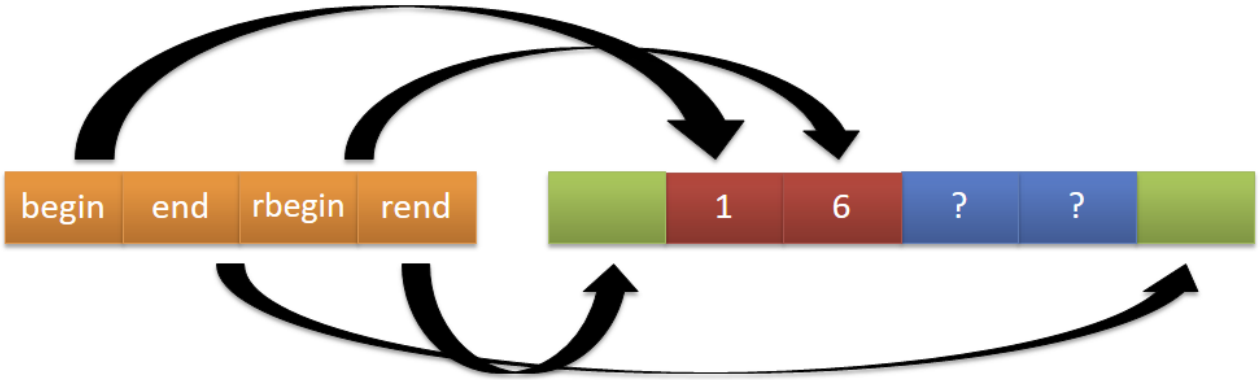
- Every container is made up of four members (types) related to iterators:
  - iterator – read/write iterator type;
  - const_iterator – read-only iterator type;
  - reverse_iterator – reverse iterator type (iterates from the end to the beginning)
  - const_reverse_iterator – as above, but read only.

Containers and iterators

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

int main()
{
    //containers
    vector<int> v;
    deque<int> d;
    list<int> l;
    //iterators
    vector<int> ::iterator it1;
    vector<int> ::const_iterator it2;
    vector<int> ::reverse_iterator it3;
    vector<int> ::const_reverse_iterator it4;

    deque<int> ::iterator it5;
    deque<int> ::const_iterator it6;
    deque<int> ::reverse_iterator it7;
    deque<int> ::const_reverse_iterator it8;

    list<int> ::iterator it9;
    list<int> ::const_iterator it10;
    list<int> ::reverse_iterator it11;
    list<int> ::const_reverse_iterator it12;

    return 0;
}
```

Initialization of iterators

- There are four methods that can do that:
  - begin()
  - end()
  - rbegin()
  - rend()



Initialization of iterators

- Each method comes in two variations – normal and const:
  - iterator begin ();
    const_iterator begin () **const**;
  - iterator end ();
    const_iterator end () **const**;
  - reverse_iterator rbegin();
    const_reverse_iterator rbegin() **const**;
  - reverse_iterator rend();
    const_reverse_iterator rend() **const**;

Initialization of iterators

- For **vector** and **deque**, these methods return random access iterators, but the **list** only supports a bidirectional iterator.
  - the begin() method returns the iterator that points to the first element of the collection;
  - the end() method returns the iterator that refers to the past-the-end-element. If a container has n elements, this value will be marked *n+1* (non-existent).
  - the rbegin() method means reverse begin – it returns the iterator that points to the last element of the collection;

Initialization of iterators

- For **vector** and **deque**, these methods return random access iterators, but the **list** only supports a bidirectional iterator.
  - the rend() method means reverse end – it returns the iterator that refers to the element before the first element of the container – this value indicates the end of the collection in reverse order.
- **Past-the-end element**
  - The *past-the-end* element is a virtual element located after the last element of the collection. It indicates the end of the collection.

Iterators usage examples – normal iterators

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

int main()
{
    //containers
    vector <int> v(10);
    deque <int> d(10);
    list <int> l(10);

    int i = 1;
    //vector
    vector<int>::iterator itV;
    for(itV = v.begin()  ; itV != v.end(); ++itV,++i)
    {
        *itV = i;
    }
    for(itV = v.begin();  itV != v.end(); ++itV)
    {
        cout << *itV << " ";
    }
    cout<<endl;
    //deque
    deque<int>::iterator itD = d.begin();
    for(itD = d.begin()  ; itD != d.end(); ++itD,++i)
    {
        *itD = i;
    }
    for( itD = d.begin() ; itD != d.end(); ++itD)
    {
        cout << *itD << " ";
    }
    cout<<endl;
    list<int>::iterator itL = l.begin();
    for( ; itL != l.end(); ++itL,++i)
    {
        *itL = i;
    }
    for( itL = l.begin() ; itL != l.end(); ++itL)
    {
        cout << *itL << " ";
    }
    cout<<endl;
    return 0;
}
```

```
Console output:
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
```

Iterators usage examples – reverse iterators

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

int main()
{
    //containers
    vector <int> v(10);
    deque <int> d(10);
    list <int> l(10);

    int i = 1;
    //vector
    vector<int>::iterator itV;
    for(itV = v.begin()  ; itV != v.end(); ++itV,++i)
    {
        *itV = i;
    }

    for(vector<int>::reverse_iterator it = v.rbegin();  it != v.rend(); ++it)
    {
        cout << *it << " ";
    }
    cout<<endl;
    //deque
    i = 1;
    deque<int>::iterator itD = d.begin();
    for(itD = d.begin()  ; itD != d.end(); ++itD,++i)
    {
        *itD = i;
    }
    for( deque<int>::reverse_iterator it = d.rbegin() ; it != d.rend(); ++it)
    {
        cout << *it << " ";
    }
    cout<<endl;
    //list
    i = 1;
    list<int>::iterator itL = l.begin();
    for( ; itL != l.end(); ++itL,++i)
    {
        *itL = i;
    }
    for(list<int>::reverse_iterator it = l.rbegin() ; it != l.rend(); ++it)
    {
        cout << *it << " ";
    }
    cout<<endl;
    return 0;
}
```

```
Console output:
10 9 8 7 6 5 4 3 2 1
10 9 8 7 6 5 4 3 2 1
10 9 8 7 6 5 4 3 2 1
```

Iterators usage examples – const iterators

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    //containers
    vector <int> v(a,a+10);
    deque <int> d(a,a+10);
    list <int> l(a,a+10);

    //vector
    for(vector<int>::const_iterator it = v.begin()  ; it != v.end(); ++it)
    {
        cout << *it << " ";
    }
    cout<<endl;
    //deque
    for(deque<int>::const_iterator it = d.begin(); it != d.end(); ++it)
    {
        cout << *it << " ";
    }
    cout<<endl;
    //list
    for(list<int>::const_iterator it = l.begin(); it != l.end(); ++it)
    {
        cout << *it << " ";
    }
    cout<<endl;
    return 0;
}
```

```
Console output:
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

Iterators usage examples

- The compiler doesn't allow anything to be assigned to an element referred to by a constant iterator.

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    //containers
    vector<int>  v(a,a+10);
    deque<int>  d(a,a+10);
    list<int>  l(a,a+10);

    vector<int> ::const_iterator it1 = v.begin();
    *it1 = *it1+1;
    deque<int> ::const_iterator it2 = d.begin();
    *it2 = *it2+1;
    list<int> ::const_iterator it3 = l.begin();
    *it3 = *it3+1;
    return 0;
}
```

Outline

1. STL sequential containers
    1. Sequence containers
    2. Iterators
    3. **Operations**

size() and max_size()

- **Name**:
  - size
- **Signature**:
  - size_type size() **const**;
- **Parameters**:
  - None.
- **Return value**:
  - The number of elements which are currently stored inside a collection.
- **Description**:
  - This method returns the number of elements which are currently stored inside a container. The size will change each time an element is added to or removed from the container.

size() and max_size()

- **Name**:
  - max_size
- **Signature**:
  - size_type max_size () **const**;
- **Parameters**:
  - None.
- **Return value**:
  - The maximum number of elements which can be held inside a container.
- **Description**:
  - The method returns the maximum physical capacity of a container. This value might depend on the STL library implementation or an operating system, and will always be constant in the same environment.

size() and max_size()

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    //containers
    vector <int> v(a,a+10);
    deque <int> d(a,a+10);
    list <int> l(a,a+10);
    cout<<"Size of vector, deque, list: "<<v.size() << " " << d.size() <<" "
        << l.size()<<endl;
    cout<<"Max size of vector, deque, list: "<<v.max_size() << " "
        << d.max_size() <<" " << l.max_size()<<endl<<endl;

    v.push_back(11);
    d.push_back(11);
    l.push_back(11);

    cout<<"Size of vector, deque, list: "<<v.size() << " " << d.size() <<" "
        << l.size()<<endl;
    cout<<"Max size of vector, deque, list: "<<v.max_size() << " "
        << d.max_size() <<" " << l.max_size()<<endl<<endl;

    v.pop_back();
    d.pop_back();
    l.pop_back();

    cout<<"Size of vector, deque, list: "<<v.size() << " " << d.size() <<" "
        << l.size()<<endl;
    cout<<"Max size of vector, deque, list: "<<v.max_size() << " "
        << d.max_size() <<" " << l.max_size()<<endl<<endl;
    return 0;
}
```

size() and max_size()

```
Console output:
Size of vector, deque, list: 10 10 10
Max size of vector, deque, list: 1073741823 1073741823 357913941

Size of vector, deque, list: 11 11 11
Max size of vector, deque, list: 1073741823 1073741823 357913941

Size of vector, deque, list: 10 10 10
Max size of vector, deque, list: 1073741823 1073741823 357913941
```

empty() and resize()

- **Name**:
  - empty
- **Signature**:
  - **bool** empty () **const**;
- **Parameters**:
  - None.
- **Return value**:
  - The method returns true if a container is empty and false otherwise.
- **Description**:
  - This method is used to indicate whether a container is empty or not. You should use this method instead of calling size() to check if the list container is empty. Using calling size() for a list might result in linear time performance for some STL implementations, instead of constant ones.

empty() and resize()

- **Name**:
  - resize
- **Signature**:
  - **void** resize (size_type sz, T c = T());
- **Parameters**:
  - sz – the new size of a container;
  - c – the value to copy in order to add new elements into a container when the new size (sz) is greater than the old size.
- **Return value**:
  - None.
- **Description**:
  - This method changes the current size of a container, either by causing it to grow or shrink. The new size is provided by the parameter sz.
  - If the new size is greater than the old size, new elements are added to the container. Those new elements are created by copying parameter c, or, if c is not provided, by copying the default value for a particular type of element.

empty() and resize()

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

int main()
{
    //containers
    vector <int> v;
    deque <int> d;
    list <int> l;
    cout<<"Size of vector, deque, list: "<<v.size() << " " << d.size() <<" "
        << l.size()<<endl;

    if (v.empty())
    {
        v.resize(10);
    }
    if (d.empty())
    {
        d.resize(10);
    }
    if (l.empty())
    {
        l.resize(10);
    }
    cout<<"Size of vector, deque, list: "<<v.size() << " " << d.size() <<" "
        << l.size()<<endl;
```

empty() and resize()

```
    if (!v.empty())
    {
        v.resize(9);
    }
    if (!d.empty())
    {
        d.resize(5);
    }
    if (!l.empty())
    {
        l.resize(0);
    }
    cout<<"Size of vector, deque, list: "<<v.size() << " " << d.size() <<" "
        << l.size()<<endl;
    return 0;
}
```

```
Console output:
Size of vector, deque, list: 0 0 0
Size of vector, deque, list: 10 10 10
Size of vector, deque, list: 9 5 0
```

vector::capacity() – vector only

- **Name**:
  - vector<T>::capacity
- **Signature**:
  - size_type capacity () **const**;
- **Parameters**: None.
- **Return value**:
  - The total amount of space for elements currently allocated to a particular vector.
- **Description**:
  - This method is present in the vector class only. **capacity** is the total number of slots inside a vector which are currently allocated. capacity is always greater than or equal to size.

vector::capacity() – vector only

```
Console output:
Output depends on implementation:
Size and capacity: 0 0
Size and capacity: 1 1
Size and capacity: 2 2
Size and capacity: 3 3
Size and capacity: 4 4
Size and capacity: 5 6
Size and capacity: 6 6
Size and capacity: 7 9
Size and capacity: 8 9
Size and capacity: 9 9
Size and capacity: 10 13
Size and capacity: 11 13
Size and capacity: 12 13
Size and capacity: 13 13
Size and capacity: 14 19
Size and capacity: 15 19
Size and capacity: 16 19
Size and capacity: 17 19
Size and capacity: 18 19
Size and capacity: 19 19
Size and capacity: 20 28
```

vector::capacity() – vector only

```
#include <vector>
#include <iostream>

using namespace std;

int main()
{
    vector <int> v;
    cout<<"Size and capacity: "<<v.size() << " " << v.capacity() <<endl;

    for(int i = 0; i < 20; ++i)
    {
        v.push_back(i);
        cout<<"Size and capacity: "<<v.size() << " " << v.capacity() <<endl;
    }
    return 0;
}
```

vector::reserve() – vector only

- **Name**:
  - vector<T>::reserve
- **Signature**:
  - **void** reserve (size_type n);
- **Parameters**:
  - n – the minimum value of capacity to be requested.
- **Return value**:
  - None.
- **Description**:
  - This method allocates additional space for elements inside a vector. If the newly requested capacity n is greater than the current capacity, reallocation is enforced.

vector::reserve() – vector only

```
#include <vector>
#include <iostream>

using namespace std;

int main()
{
    vector <int> v;
    cout<<"Size and capacity: "<<v.size() << " " << v.capacity() <<endl;
    v.reserve(15);
    cout<<"Size and capacity: "<<v.size() << " " << v.capacity() <<endl;
    cout<<"Adding elements"<<endl;
    for(int i = 0; i < 10; ++i)
    {
        v.push_back(i);
        cout<<"Size and capacity: "<<v.size() << " " << v.capacity() <<endl;
    }
    cout<<"Trying to shrink..."<<endl;
    v.reserve(10);
    cout<<"Size and capacity: "<<v.size() << " " << v.capacity() <<endl;
    return 0;
}
```

vector::reserve() – vector only

```
Console output:
Size and capacity: 0 0
Size and capacity: 0 15
Adding elements
Size and capacity: 1 15
Size and capacity: 2 15
Size and capacity: 3 15
Size and capacity: 4 15
Size and capacity: 5 15
Size and capacity: 6 15
Size and capacity: 7 15
Size and capacity: 8 15
Size and capacity: 9 15
Size and capacity: 10 15
Trying to shrink...
Size and capacity: 10 15
```

back() and front()

- **Name**:
  - front
- **Signature**:
  - reference front ( );
    const_reference front ( ) **const**;
- **Parameters**:
  - None.
- **Return value**:
  - A reference to the first element in a container.
- **Description**:
  - This method returns a reference to the first element in a container. The reference might be normal or constant – it all depends on the calling context.

back() and front()

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    I it;
    for(it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}

int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    //containers
    vector <int>  v(a,a+10);
    deque <int>  d(a,a+10);
    list <int>  l(a,a+10);

    cout<<"Size of vector, deque, list: "<<v.size() << " " << d.size()
        <<" " << l.size()<<endl;
    cout<<"Values at front (vector, deque, list): "<< v.front() << " " <<d.front()
        << " " <<l.front() <<endl;
    cout<<"Size of vector, deque, list: "<<v.size() << " " << d.size()
        <<" " << l.size()<<endl;

    v.front() = 100;
    d.front() = 101;
    l.front() = 102;
    print(v.begin(), v.end());
    print(d.begin(), d.end());
    print(l.begin(), l.end());
    return 0;
}
```

back() and front()

```
Console output:
Size of vector, deque, list: 10 10 10
Values at front (vector, deque, list): 1 1 1
Size of vector, deque, list: 10 10 10
100 2 3 4 5 6 7 8 9 10
101 2 3 4 5 6 7 8 9 10
102 2 3 4 5 6 7 8 9 10
```

back() and front()

- **Name**:
  - back
- **Signature**:
  - reference back ( );
    const_reference back ( ) **const**;
- **Parameters**:
  - None.
- **Return value**:
  - A reference to the last element in a container.
- **Description**:
  - This method returns a reference to the last element in a container. The reference might be normal or constant – it all depends on the calling context.

back() and front()

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    I it;
    for(it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}

int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    //containers
    vector <int>  v(a,a+10);
    deque <int>  d(a,a+10);
    list <int> l(a,a+10);

    cout<<"Size of vector, deque, list: "<<v.size() << " " << d.size()
        <<" " << l.size()<<endl;
    cout<<"Values at back (vector, deque, list): "<< v.back() << " " <<d.back()
        << " " <<l.back() <<endl;
    cout<<"Size of vector, deque, list: "<<v.size() << " " << d.size()
        <<" " << l.size()<<endl;

    v.back() = 100;
    d.back() = 101;
    l.back() = 102;
    print(v.begin(), v.end());
    print(d.begin(), d.end());
    print(l.begin(), l.end());
    return 0;
}
```

back() and front()

```
Console output:
Size of vector, deque, list: 10 10 10
Values at back (vector, deque, list): 10 10 10
Size of vector, deque, list: 10 10 10
1 2 3 4 5 6 7 8 9 100
1 2 3 4 5 6 7 8 9 101
1 2 3 4 5 6 7 8 9 102
```

operator[] and at() – vector and deque only

- **Name**:
  - operator[] (vector and deque only)
- **Signature**:
  - reference **operator**[] (size_type n);
    const_reference **operator**[] (size_type n) const
- **Parameters**:
  - n – the index of the element to access.
- **Return value**:
  - A reference to the element of index n.
- **Description**:
  - Operator [] allows containers (vector and deque in this case) to be treated in a similar way to arrays.

operator[] and at() – vector and deque only

```
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

template <class C>
void print (const C & container)
{
    for(unsigned i = 0; i < container.size(); ++i)
    {
        cout<< container[i] << " ";
    }
    cout<<endl;
}

int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    //containers
    vector <int> v(10);
    deque <int> d(10);

    for(unsigned i = 0; i < 10; ++i)
    {
        d[i] = a[i];
        v[i] = a[i];
    }
    print(v);
    print(d);
    cout<<"Accessing out of range element:\n";
    cout<<v[10]<<" " <<d[10]<<endl;

    return 0;
}
```

```
Console output (g++)
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
Accessing out of range element:
0 0
```

operator[] and at() – vector and deque only

- **Name**:
  - at (vector and deque only)
- **Signature**:
  - reference **at** (size_type n);
    const_reference **at** (size_type n) const
- **Parameters**:
  - n – the index of the element to be accessed.
- **Return value**:
  - A reference to the element of index n.
- **Description**:
  - The method at() is used to retrieve an element from the STL container (vector and deque). It retrieves the value stored under the index n. at() is very similar in its behavior to operator[]. The only difference is that at() performs a range check on parameter n, and if n is out of range, an out_of_range exception is thrown.

operator[] and at() – vector and deque only

```
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

template <class C>
void print (const C & container)
{
    for(unsigned i = 0; i < container.size(); ++i)
    {
        cout<< container.at(i) << " ";
    }
    cout<<endl;
}
```

```
int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    //containers
    vector <int>  v(10);
    deque <int>  d(10);

    for(unsigned i = 0; i < 10; ++i)
    {
        d.at(i) = a[i];
        v.at(i) = a[i];
    }
    print(v);
    print(d);
    cout<<"Accessing out of range element:\n";
    try
    {
        cout<<v.at(11)<<endl;
    }
    catch (out_of_range & ex)
    {
        cout<< ex.what()<<endl;
    }

    try
    {
        cout<<d.at(11)<<endl;
    }
    catch (out_of_range & ex)
    {
        cout<< ex.what()<<endl;
    }


    return 0;
}
```

```
Console output:
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
Accessing out of range element:
invalid vector subscript
invalid deque subscript
```

assign()

- **Name**:
  - assign
- **Signature**:
  - **template** <**class** InputIterator>
    void assign (InputIterator first, InputIterator last);
    void assign (size_type n, const T& u);
- **Parameters**:
  - first, last – the input iterators which provide a collection of input elements. The assign method will copy all the elements from this range, including first and excluding last. Because first and last are of the InputIterator type, virtually any type of iterator can be used in the call;
  - n – the number of times the value u will be copied to fill the container;
  - u – the value to be copied.
- **Return value**:
  - None.
- **Description**:
  - This method assigns new values to an already existing container. The whole old content of the container is dropped and deleted.

assign()

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    I it;
    for(it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}
```

```
int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    //containers
    vector <int> v(a,a+5);
    deque <int> d(a,a+5);
    list <int> l(a,a+5);

    print(v.begin(), v.end());
    print(d.begin(), d.end());
    print(l.begin(), l.end());

    cout<<"Assigning a new content:\n";
    v.assign(a, a+10);
    d.assign(a, a+10);
    l.assign(a, a+10);
    print(v.begin(), v.end());
    print(d.begin(), d.end());
    print(l.begin(), l.end());

    cout<<"Assigning a new content II:\n";
    v.assign(3, 100);
    d.assign(3, 1000);
    l.assign(3, 10000);
    print(v.begin(), v.end());
    print(d.begin(), d.end());
    print(l.begin(), l.end());
    return 0;
}
```

```
Console output:
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
Assigning a new content:
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
Assigning a new content II:
100 100 100
1000 1000 1000
10000 10000 10000
```

insert()

- **Name**:
  - insert
- **Signature**:
  - iterator insert (iterator position, **const** T& x);
    void insert (iterator position, size_type n, **const** T& x);
    template <**class** InputIterator>
    void insert (iterator position, InputIterator first, InputIterator last);
- **Parameters**:
  - position – the position in the container at which the insertion of an element (or elements) is to be performed. For deque and vector, this is RandomAccessIterator, while in the case of a list, BidirectionalIterator is used;
  - x – the value to be inserted;
  - n – the number of x values to be inserted;
  - first, last – the iterators specifying the range of elements to be inserted into the container. As usual, the range includes first and excludes last.
- **Return value**:
  - The first version of this method returns an iterator to a newly inserted object if the insertion is successful. Other versions do not return anything.

insert()

- **Description**:
  - The method insert() performs an insertion into a container. There are three variants of this method, as stated in the signature section. Inserting an element into a container will cause the container to grow. This leads to different consequences, depending on the type of collection:
  - vector – when an increase in size causes it to reallocate (not enough capacity left) all iterators, references and pointers will be invalidated;
  - deque – all iterators will be invalidated, references also, unless insertion at the beginning or end takes place;
  - list – the iterators and references remain.

insert()

```
#include <vector>
#include <iostream>

using namespace std;

template <class I>
void print (const I & start, const I & end)
{
    I it;
    for(it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}

int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    vector <int> v(a,a+10);

    vector<int>::iterator it = v.insert(v.begin()+5, 100);
    print(v.begin(), v.end());
    cout<<"Inserted element: "<<*it<<endl;
    cout <<"Size: "<<v.size()<<endl;

    vector <int> v2;
    v2.insert(v2.begin(), v.rbegin(), v.rend());
    print(v2.begin(), v2.end());

    vector <int> v3(v.begin(), v.begin()+5);
    v3.insert(v3.end(),3,100);

    print(v3.begin(), v3.end());

    return 0;
}
```

```
Console output:
1 2 3 4 5 100 6 7 8 9 10
Inserted element: 100
Size: 11
10 9 8 7 6 100 5 4 3 2 1
1 2 3 4 5 100 100 100
```

insert()

```
using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    I it;
    for(it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}

int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    deque <int> d1(a,a+10);

    deque<int>::iterator it = d1.insert(d1.begin()+5, 100);
    print(d1.begin(), d1.end());
    cout<<"Inserted element: "<<*it<<endl;
    cout <<"Size: "<<d1.size()<<endl;

    deque <int> d2;
    d2.insert(d2.begin(), d1.rbegin(), d1.rend());
    print(d2.begin(), d2.end());

    deque <int> d3(d1.begin(), d1.begin()+5);
    d3.insert(d3.end(),3,100);

    print(d3.begin(), d3.end());

    return 0;
}
```

```
Console output:
1 2 3 4 5 100 6 7 8 9 10
Inserted element: 100
Size: 11
10 9 8 7 6 100 5 4 3 2 1
1 2 3 4 5 100 100 100
```

insert()

```
#include <list>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    I it;
    for(it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}

int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    list <int> l1(a,a+10);

    list<int>::iterator it = l1.insert(l1.begin(), 100);
    print(l1.begin(), l1.end());
    cout<<"Inserted element: "<<*it<<endl;
    cout <<"Size: "<<l1.size()<<endl;

    list <int> l2;
    l2.insert(l2.begin(), l1.rbegin(), l1.rend());
    print(l2.begin(), l2.end());

    list<int>::iterator it1 = l1.begin();
    //shifting 5 places
    for(int i = 0; i < 5; ++i)
    {
        ++it1;
    }
    list <int> l3(l1.begin(), it1);
    l3.insert(l3.end(),3,100);

    print(l3.begin(), l3.end());

    return 0;
}
```

```
Console output:
100 1 2 3 4 5 6 7 8 9 10
Inserted element: 100
Size: 11
10 9 8 7 6 5 4 3 2 1 100
100 1 2 3 4 100 100 100
```

erase()

- **Name**:
  - erase
- **Signature**:
  - iterator erase (iterator position);
    iterator erase (iterator first, iterator last);
- **Parameters**:
  - position – the iterator pointing to the element to be erased;
  - first, last – the iterators which specify the range of elements to be erased. As usual, the range includes first and excludes last.
- **Return value**:
  - An iterator to the first element after the last removed element, or end if the operation removes the last element in the collection.
- **Description**:
  - This function removes an element or a range of elements from a collection.

erase()

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    I it;
    for(it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}
```

```
int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    vector <int> v(a,a+10);
    deque <int> d(a,a+10);
    list <int> l(a,a+10);

    print(v.begin(), v.end());
    print(d.begin(), d.end());
    print(l.begin(), l.end());

    cout<<"Ereasing elements:\n";
    v.erase(v.begin()+3);
    d.erase(d.begin()+3);
    //no random access iterator
    list<int>::iterator it= l.begin();
    ++it; ++it; ++it;
    it = l.erase(it);
    print(v.begin(), v.end());
    print(d.begin(), d.end());
    print(l.begin(), l.end());

    cout<<"Ereasing elements:\n";
    v.erase(v.begin()+3, v.end());
    d.erase(d.begin()+3, d.end());
    l.erase(it, l.end());
    print(v.begin(), v.end());
    print(d.begin(), d.end());
    print(l.begin(), l.end());
    return 0;
}
```

```
Console output:
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
Ereasing elements:
1 2 3 5 6 7 8 9 10
1 2 3 5 6 7 8 9 10
1 2 3 5 6 7 8 9 10
Ereasing elements:
1 2 3
1 2 3
1 2 3
```

swap()

- **Name**:
  - swap
- **Signature**:
  - **void** vector::swap (vector<T,Allocator>& vec);
    void deque::swap (deque<T,Allocator>& dqe);
    void list::swap (list<T,Allocator>& lst);
- **Parameters**:
  - vec, deq, lst – another collection of the same type as this one.
- **Return value**:
  - None.
- **Description**:
  - This method swaps the entire content between two collections of the same type (list <–> list, vector <–> vector, deque <–> deque).

swap()

```
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    I it;
    for(it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}
```

```
int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    vector <int> v1(a,a+5);
    deque <int> d1(a,a+5);
    list <int> l1(a,a+5);

    vector<int> v2(a+5,a+10);
    deque<int> d2(a+5,a+10);
    list<int>  l2(a+5,a+10);

    print(v1.begin(), v1.end());
    print(v2.begin(), v2.end());
    print(d1.begin(), d1.end());
    print(d2.begin(), d2.end());
    print(l1.begin(), l1.end());
    print(l2.begin(), l2.end());

    cout<<"Swapping elements:\n";
    v1.swap(v2);
    d1.swap(d2);
    l1.swap(l2);
    print(v1.begin(), v1.end());
    print(v2.begin(), v2.end());
    print(d1.begin(), d1.end());
    print(d2.begin(), d2.end());
    print(l1.begin(), l1.end());
    print(l2.begin(), l2.end());

    return 0;
}
```

```
Console output:
1 2 3 4 5
6 7 8 9 10
1 2 3 4 5
6 7 8 9 10
1 2 3 4 5
6 7 8 9 10
Swapping elements:
6 7 8 9 10
1 2 3 4 5
6 7 8 9 10
1 2 3 4 5
6 7 8 9 10
1 2 3 4 5
```

clear()

- **Name**:
  - clear
- **Signature**:
  - void clear ( );
- **Parameters**:
  - None.
- **Return value**:
  - None.
- **Description**:
  - This function removes all the elements from the collection and sets its size to 0. During the removal, the destructors are called.

clear()

```
#include <list>
#include <vector>
#include <deque>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    for(I it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}
```

```
int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    vector <int> v(a,a+10);
    deque <int> d(a,a+10);
    list <int> l(a,a+10);

    print(v.begin(), v.end());
    print(d.begin(), d.end());
    print(l.begin(), l.end());

    cout<<"Clearing collections:\n";
    v.clear();
    d.clear();
    l.clear();

    v.push_back(100);
    d.push_back(100);
    l.push_back(100);
    print(v.begin(), v.end());
    print(d.begin(), d.end());
    print(l.begin(), l.end());

    return 0;
}
```

```
Console output:
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
Clearing collections:
100
100
100
```

push_back() and pop_back()

- **Name**:
  - push_back
- **Signature**:
  - **void** push_back (**const** T& x);
- **Parameters**:
  - x – the value which will be used to create a new element (by copying) inside a container.
- **Return value**: None.
- **Description**:
  - The function push_back() adds a new value to a container. The value is added at the end (the back) of the container, and increases the size of the container by one. Different types of containers react differently:
  - if a vector has enough *capacity*, the item is just added to it, no reallocation is performed, and all obtained iterators remain valid;
  - if there is not enough *capacity* left, a reallocation is performed, which invalidates all iterators;

push_back() and pop_back()

- **Name**:
  - pop_back
- **Signature**:
  - void pop_back ( );
- **Parameters**:
  - None.
- **Return value**:
  - None.
- **Description**:
  - This function removes an element from the tail of the container. Basically, it is the opposite method to push_back().

push_back() and pop_back()

```
#include <vector>
#include <deque>
#include <list>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    for(I it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}
```

```
int main()
{
    vector <int> v;
    deque <int> d;
    list <int> l;

    for(unsigned i = 0; i < 10; ++i)
    {
        v.push_back(i);
        d.push_back(i);
        l.push_back(i);
    }
    cout<<"Vector: ";    print(v.begin(), v.end());
    cout<<"Deque:  ";    print(d.begin(), d.end());
    cout<<"List:   ";    print(l.begin(), l.end());

    for(unsigned i = 0; i < 5; ++i)
    {
        v.pop_back();
        d.pop_back();
        l.pop_back();
    }

    cout<<"Vector: ";    print(v.begin(), v.end());
    cout<<"Deque:  ";    print(d.begin(), d.end());
    cout<<"List:   ";    print(l.begin(), l.end());

    return 0;
}
```

Console output:

```
Vector: 0 1 2 3 4 5 6 7 8 9
Deque:  0 1 2 3 4 5 6 7 8 9
List:   0 1 2 3 4 5 6 7 8 9
Vector: 0 1 2 3 4
Deque:  0 1 2 3 4
List:   0 1 2 3 4
```

push_front() and pop_front()

- **Name**:
  - push_front(deque and list only)
- **Signature**:
  - **void** push_front (**const** T& x);
- **Parameters:**
  - x – the value which will be used to create a new element (by copying) inside a container.
- **Return value**:
  - None.
- **Description:**
  - The function push_front() adds a new value to a container. The value is added at the beginning (the front) of the container, and increases the size of the container by one.

push_front() and pop_front()

- **Name**:
  - pop_front
- **Signature**:
  - **void** pop_front ( );
- **Parameters**:
  - None.
- **Return value**:
  - None.
- **Description**:
  - This function removes an element from the beginning of the container. Basically, it's the opposite method to push_front(). During element removal, its destructor is called, and the container size is reduced by one.

push_front() and pop_front()

```
#include <deque>
#include <list>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    for(I it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}

int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    deque <int> d;
    list <int> l;

    for(unsigned i = 0; i < 10; ++i)
    {
        d.push_front(i);
        l.push_front(i);
    }
    cout<<"Deque:  ";    print(d.begin(), d.end());
    cout<<"List:   ";    print(l.begin(), l.end());

    for(unsigned i = 0; i < 5; ++i)
    {
        d.pop_front();
        l.pop_front();
    }
    cout<<"Deque:  ";    print(d.begin(), d.end());
    cout<<"List:   ";    print(l.begin(), l.end());

    return 0;
}
```

```
Console output:
Deque:  9 8 7 6 5 4 3 2 1 0
List:   9 8 7 6 5 4 3 2 1 0
Deque:  4 3 2 1 0
List:   4 3 2 1 0
```

splice() – list only

- **Name:**
  - splice (list)
- **Signature:**
  - **void** splice (iterator position, list<T,Allocator>& x);
    void splice (iterator position, list<T,Allocator>& x, iterator i);
    void splice (iterator position, list<T,Allocator>& x, iterator first, iterator last);
- **Parameters:**
  - position – the position in the calling list where the elements will be inserted;
  - x – the list from which the elements will be moved to the calling list;
  - i – the iterator to a single element from the source list, which will be moved to the calling list;
  - first, last – the iterators which define the range of elements to be moved from the source list to the destination. The range includes first and excludes last.

splice() – list only

- **Return value**:
  - None.
- **Description**:
  - This method moves elements from a list specified as parameter x, and inserts them into the list container which calls the method. The target list size increases by the number of elements moved, while the source list size decreases accordingly. There are three versions of this method:
    - a method which moves the whole content of the source container;
    - a method which moves one, and only one, element specified by the iterator;
    - a method which moves a range of elements specified by the iterators.

splice() – list only

```
#include <list>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    for(I it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}
```

```
int main()
{
    int a[]={1,2,3,4,5};
    int b[]={6,7,8,9,10};
    int c[]={11,12,13,14,15};
    list <int> l1(a,a+5);
    list <int> l2(b, b+5);
    list <int> l3(c, c+5);

    l2.splice(l2.end(),l3);
    print(l2.begin(), l2.end());
    cout<<"Size of source list l3: "<< l3.size()<<endl;

    //moving one element - '15'
    list<int>::iterator it = l2.begin();
    advance(it,9);
    l1.splice(l1.end(),l2,it);
    print(l1.begin(), l1.end());
    cout<<"Size of source list l2: "<< l2.size()<<endl;

    //moving range of elements
    it = l1.end();
    advance(it,-1);
    l1.splice(it,l2,l2.begin(), l2.end());
    print(l1.begin(), l1.end());
    cout<<"Size of source list l2: "<< l2.size()<<endl;

    return 0;
}
```

```
Console output:
6 7 8 9 10 11 12 13 14 15
Size of source list l3: 0
1 2 3 4 5 15
Size of source list l2: 9
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Size of source list l2: 0
```

remove() and remove_if() – list only

- **Name**:
  - remove (list)
- **Signature**:
  - **void** remove (**const** T& value);
- **Parameters**:
  - value – the value of the element to be removed from the list. It's the same type as that used during the list declaration.
- **Return value:**
  - None.
- **Description:**
  - This function removes from the list all the elements equal to the values provided as the parameters. During the removal, the destructors are called.

remove() and remove_if() – list only

- **Name:**
  - remove_if (list)
- **Signature:**
  - template <class Predicate>
    void remove_if (Predicate pred);
- **Parameters:**
  - pred – a unary predicate (one argument function, or function object) which takes an argument of the same type as the elements of the list. The predicate should return true for elements which are to be removed, and false for all others.
- **Return value**:
  - None.
- **Description:**
  - The function remove_if() performs a conditional object deletion. The method calls the provided predicate for every element stored inside the list. If the predicate returns true, the element is eligible for removal.

remove() and remove_if() – list only

```
#include <list>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    for(I it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}

int main()
{
    int a[]={1,2,1,3,2,3,4,3,4,7,8,9,6,6,5,8,9,10};

    list <int> l1(a,a+18);
    print(l1.begin(), l1.end());
    //remove all '1'
    l1.remove(1);
    cout<<"All '1' deleted"<<endl;
    print(l1.begin(), l1.end());
    //remove all '2'
    l1.remove(2);
    cout<<"All '2' deleted"<<endl;
    print(l1.begin(), l1.end());
    //remove all '3'
    l1.remove(3);
    cout<<"All '3' deleted"<<endl;
    print(l1.begin(), l1.end());

    return 0;
}
```

```
Console output:
1 2 1 3 2 3 4 3 4 7 8 9 6 6 5 8 9 10
All '1' deleted
2 3 2 3 4 3 4 7 8 9 6 6 5 8 9 10
All '2' deleted
3 3 4 3 4 7 8 9 6 6 5 8 9 10
All '3' deleted
4 4 7 8 9 6 6 5 8 9 10
```

remove() and remove_if() – list only

```
#include <list>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    for(I it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}
struct DeleteOdd
{
    bool operator ()(int value)
    {
        if (value % 2 > 0 )
        {
            return true;
        }
        return false;
    }
};
bool deleteEven(int value)
{
    if (value % 2 == 0 )
    {
        return true;
    }
    return false;
}
```

```
int main()
{
    int a[]={1,2,1,3,2,3,4,3,4,7,8,9,6,6,5,8,9,10};

    list <int> l1(a,a+18);
    list <int> l2(a,a+18);
    print(l1.begin(), l1.end());
    //remove odd numbers
    l1.remove_if(DeleteOdd());
    cout<<"All odd numbers have been deleted"<<endl;
    print(l1.begin(), l1.end());
    //remove even numbers
    l2.remove_if(deleteEven);
    cout<<"All even numbers have been deleted"<<endl;
    print(l2.begin(), l2.end());

    return 0;
}
```

```
Console output:
1 2 1 3 2 3 4 3 4 7 8 9 6 6 5 8 9 10
All odd numbers have been deleted
2 2 4 4 8 6 6 8 10
All even numbers have been deleted
1 1 3 3 3 7 9 5 9
```

sort() – list only

- **Name:**
  - sort (list)
- **Signature:**
  - void sort ( );
    template <class Compare>
    void sort (Compare comp);
- **Parameters:**
  - comp – the binary predicate used to compare pairs of elements in order to ensure a proper sort order. It takes arguments of the same type as the elements of the list.
- **Return value**:
  - None.
- **Description**:
  - This method performs the sorting of elements in a lexicographic order – from lowest to highest.

sort() – list only

```
#include <list>
#include <iostream>

using namespace std;

template<class I>
void print (const I & start, const I & end)
{
    for(I it = start; it != end; ++it)
    {
        cout<< *it << " ";
    }
    cout<<endl;
}

bool compare(int v1, int v2)
{
    if (v1 > v2)
    {
        return true;
    }
    return false;
}
```

```
int main()
{
    int a[]={1,2,1,3,2,3,4,7,8,9,6,5,8,9,10};

    list <int> l1(a,a+15);

    print(l1.begin(), l1.end());

    cout<<"Sorting - ascending"<<endl;
    l1.sort();
    print(l1.begin(), l1.end());

    cout<<"Sorting - descending"<<endl;
    l1.sort(compare);
    print(l1.begin(), l1.end());

    return 0;
}
```

```
Console output:
1 2 1 3 2 3 4 7 8 9 6 5 8 9 10
Sorting - ascending
1 1 2 2 3 3 4 5 6 7 8 8 9 9 10
Sorting - descending
10 9 9 8 8 7 6 5 4 3 3 2 2 1 1
```