C++ List

Constructors

Syntax:

```
   list();
   list( const list& c );
   explicit list( size_type num, const TYPE& val = TYPE() );
   list( input_iterator start, input_iterator end );
```

The default list constructor takes no arguments, creates a new instance of that list.

The second constructor is a default copy constructor that can be used to create a new list that is a copy of the given list c.

The third constructor creates a list with space for num objects. If val is specified, each of those objects will be given that value. For example, the following code creates a list consisting of five copies of the integer 42:

```
  list <int> l1( 5, 42 );
```

The last constructor creates a list that is initialized to contain the elements between start and end.

Operators

Syntax:

```
   list& operator=(const list& c2);
   bool operator==(const list& c1, const list& c2);
   bool operator!=(const list& c1, const list& c2);
   bool operator<(const list& c1, const list& c2);
   bool operator>(const list& c1, const list& c2);
   bool operator<=(const list& c1, const list& c2);
   bool operator>=(const list& c1, const list& c2);
```

All of the C++ containers can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Performing a comparison or assigning one list to another takes linear time.

Two lists are equal if:

- Their size is the same, and
- Each member in location i in one list is equal to the the member in location i in the other list.

Comparisons among lists are done lexicographically.

assign

Syntax:

```
   void assign( size_type num, const TYPE& val );
   void assign( input_iterator start, input_iterator end );
```

The assign() function either gives the current list the values from start to end, or gives it num copies of val.

This function will destroy the previous contents of the list.

For example, the following code uses assign() to put 10 integers of 42 into a list:

```
 list<int> l;
l.assign( 10, 42 );
for( list<int>::iterator it = l.begin(); it != l.end(); it++ ) {
 cout << *it << " ";
}
cout << endl;
```

back

Syntax:

```
   TYPE& back();
   const TYPE& back() const;
```

The back() function returns a reference to the last element in the list.

begin

Syntax:

```
   iterator begin();
   const_iterator begin() const;
```

The function begin() returns an iterator to the first element of the list. begin() should run in constant time.

clear

Syntax:

```
void clear();
```

The function clear() deletes all of the elements in the list. clear() runs in linear time.

empty

Syntax:

```
bool empty() const;
```

The empty() function returns true if the list has no elements, false otherwise.

end

Syntax:

```
   iterator end();
   const_iterator end() const;
```

The end() function returns an iterator just past the end of the list.

Note that before you can access the last element of the list using an iterator that you get from a call to end(), you'll have to decrement the iterator first.

For example, the following code uses begin() and end() to iterate through all of the members of a list:

```
 list<int> v1( 5, 789 );
list<int>::iterator it;
for( it = v1.begin(); it != v1.end(); ++it ) {
  cout << *it << endl;
}
```

The iterator is initialized with a call to begin(). After the body of the loop has been executed, the iterator is incremented and tested to see if it is equal to the result of calling end(). Since end() returns an iterator pointing to an element just after the last element of the vector, the loop will only stop once all of the elements of the vector have been displayed.

erase

Syntax:

```
iterator erase( iterator loc );
iterator erase( iterator start, iterator end );
```

The erase method either deletes the element at location loc, or deletes the elements between start and end (including start but not including end). The return value is the element after the last element erased.

The first version of erase (the version that deletes a single element at location loc) runs in constant time. The multiple-element version of erase always takes linear time.

Lists have the important property that insertion and splicing do not invalidate iterators to list elements, and that even removal invalidates only the iterators that point to the elements that are removed.

The ordering of iterators may be changed (that is, list<T>::iterator might have a different predecessor or successor after a list operation than it did before), but the iterators themselves will not be invalidated or made to point to different elements unless that invalidation or mutation is explicit.

front

Syntax:

```
TYPE& front();
const TYPE& front() const;
```

The front() function returns a reference to the first element of the list, and runs in constant time.

insert

Syntax:

```
iterator insert( iterator loc, const TYPE& val );
void insert( iterator loc, size_type num, const TYPE& val );
template<TYPE> void insert( iterator loc, input_iterator start, input_iterator end );
```

The insert() function either:

- inserts val before loc, returning an iterator to the element inserted,
- inserts num copies of val before loc, or
- inserts the elements from start to end before loc.

max_size

Syntax:

```
 size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the list can hold. The max_size() function should not be confused with the size function, which return the number of elements currently in the list.

merge

Syntax:

```
   void merge( list& other );
   void merge( list& other, BinPred compfunction );
```

The function merge() merges all elements of other into *this, making other empty. The resulting list is ordered with respect to the < operator. If compfunction is specified, then it is used as the comparison function for the lists instead of <.

pop_back

Syntax:

```
 void pop_back();
```

The pop_back() function removes the last element of the list.

pop_front

Syntax:

```
 void pop_front();
```

The function pop_front() removes the first element of the list.

push_back

Syntax:

```
  void push_back( const TYPE& val );
```

The push_back() function appends val to the end of the list. For example, the following code puts 10 integers into a list:

```
 list<int> the_list;
for( int i = 0; i < 10; i++ )
  the_list.push_back( i );
```

---

push_front

Syntax:

```
void push_front( const TYPE& val );
```

The push_front() function inserts val at the beginning of list.

---

rbegin

Syntax:

```
 reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current list (the position of the last element).

---

remove

Syntax:

```
 void remove( const TYPE &val );
```

The function remove() removes all elements that are equal to val from the list. For example, the following code creates a list of the first 10 characters of the alphabet, then uses remove() to remove the letter 'E' from the list:

```
 // Create a list that has the first 10 letters of the alphabet
list<char> charList;
for( int i=0; i < 10; i++ )
  charList.push_front( i + 65 );
// Remove all instances of 'E'
charList.remove( 'E' );
```

---

remove_if

Syntax:

```
void remove_if( UnPred pr );
```

The remove_if() function removes all elements from the list for which the unary predicate pr is true.

---

rend

Syntax:

```
  reverse_iterator rend();
  const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to an element just before the first element of the current list.

---

resize

Syntax:

```
void resize( size_type size, TYPE val = TYPE() );
```

The resize method changes the size of the list to size. If val is specified then any newly-created elements will be initialized to have a value of val.

---

reverse

Syntax:

```
void reverse();
```

The function reverse() reverses the list, and takes linear time.

---

size

Syntax:

```
size_type size() const;
```

The size() function returns the number of elements in the current list.

---

sort

Syntax:

```
  void sort();
    void sort( BinPred p );
```

The sort() function is used to sort lists into ascending order. Ordering is done via the < operator, unless p is specified, in which case it is used to determine if an element is less than another.

---

splice

Syntax:

```
    void splice( iterator pos, list& lst );
    void splice( iterator pos, list& lst, iterator del );
    void splice( iterator pos, list& lst, iterator start, iterator end );
```

The splice function moves one or more items from lst right before location pos. The first overloading moves all items to lst, the second moves just the item at del, and the third moves all items in the range inclusive of start and exclusive of end.

splice simply moves elements from one list to another, and doesn't actually do any copying or deleting. Because of this, splice runs in constant time except for the third overloading which needs no more than linear time in the case that lst is not the same as this. However, if size is linear complexity then splice is constant time for all three.

---

swap

Syntax:

```
  void swap( list& from );
```

The swap() function exchanges the elements of the current list with those of from. This function operates in constant time.

For example, the following code uses the swap() function to exchange the values of two lists:

```
     list<string> l1;
    l1.push_back("I'm in l1!");

    list<string> l2;
    l2.push_back("And I'm in l2!");

    l1.swap(l2);
```

---

unique

Syntax:

```
   void unique();
   void unique( BinPred pr );
```

The function unique() removes all consecutive duplicate elements from the list.

Note that only consecutive duplicates are removed, which may require that you sort() the list first.

Equality is tested using the == operator, unless pr is specified as a replacement. The ordering of the elements in a list should not change after a call to unique().