GAME2001 Data Structures and Algorithms

Fall 2020

GEORGE BROWN COLLEGE

Week 5

Link Lists

Link Lists

- Array
  - deletion is slow,
  - searching in unordered arrays is slow,
  - insertion into ordered arrays is slow,
  - growing and shrinking arrays is slow

- Link lists
  - solve the array’s disadvantages such as the ability to quickly grow and shrink as well as fast removal

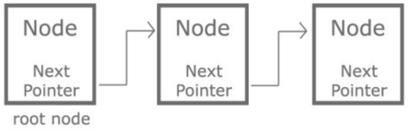
Link Lists

- a type of data structure that allows elements of the list to be linked to one another to form a dynamic chain
- each item in the list is referred to as a node (also called a link)
- start off with a root node and add elements to that node to form the chain

Link Lists

- each node has
  - a pointer to the next item in the chain
  - pointers to the item that comes before it and
  - even pointers to the start and end of the list

Visual Example of a Link List



Nodes are connected by pointers.

Link Lists

- have very fast insertions and expansion
- keys to link lists is the use of pointers

- Circular link list
  - The last nodes next pointer points to the root node
  - The roots previous node points to the last node
  - A chain that is completely circular

Link Lists

- elements of a link list are connected by pointers that can be allocated at any time
- the data do not have to exist side by side in the computer's memory
  - means random access is not possible because you can't use array indexes to access any elements you want
  - you have to start at the root and traverse through the list

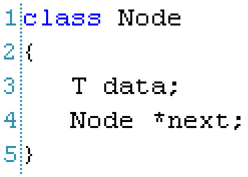
Link Lists

- normally made up of three parts
  - the node,
  - an iterator,
  - the link list itself

Link Lists

- The node’s definition can be a structure or a class that has some kind of data member and a self-referencing pointer, which is a pointer to an object of the same data type as itself

- Singly linked list
  - consists of nodes that go in one direction and is specified by the next pointer



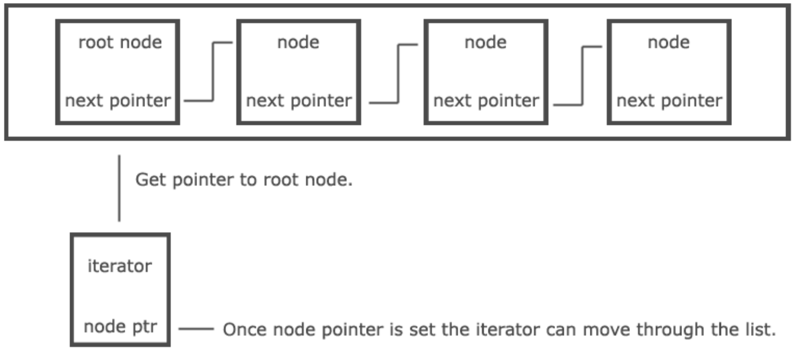
```
class Node
{
    T data;
    Node *next;
}
```

Link Lists

- Iterators
  - Will provide a way to access the elements of the linked list since we don’t have random access
  - points to an element within the list
  - used to access the data of the element
  - used to traverse through the remaining elements of the list

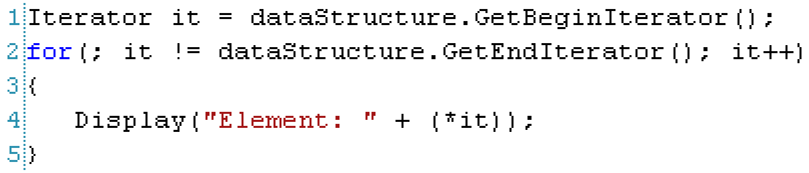
Link Lists

- Iterators
  - internally stores a pointer to a node
  - whenever an operation is applied to the iterator, it can be transferred to the node pointer
    - to move to the next element using an iterator, we could set the iterator’s node pointer to its next pointer


root node
next pointer
node
next pointer
node
next pointer
node
next pointer
Get pointer to root node.
iterator
node ptr
Once node pointer is set the iterator can move through the list.

Link Lists

- Iterators
  - Can use overloaded operators to traverse the iterator
  - Pointer dereferencing can be used to access the actual data



```
Iterator it = dataStructure.GetBeginIterator();
for(; it != dataStructure.GetEndIterator(); it++)
{
    Display("Element: " + (*it));
}
```

Singly Linked List

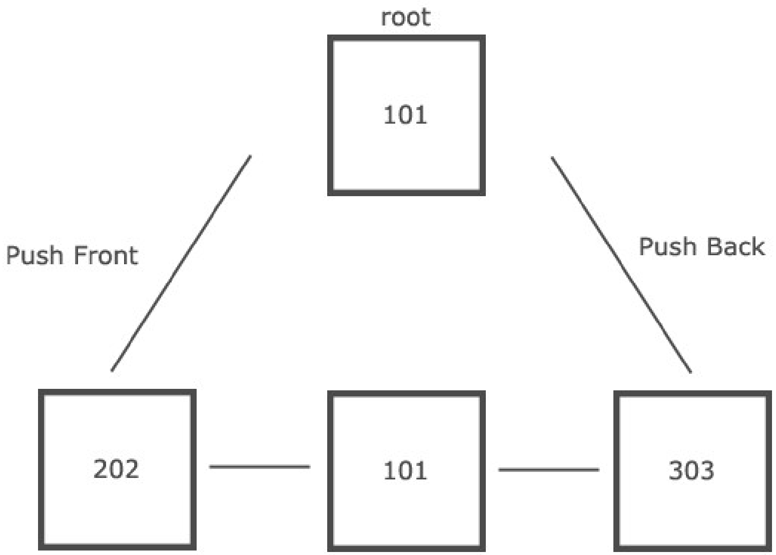
- list with nodes that go in one direction
  - – the iterator
    - a structure that will be used to access and traverse through the link list data structure
  - – the node
    - never directly used and only exists in the link list
  - – the link list
    - the container class for everything

Singly Linked List

- Example

Double-Ended Link List

- allows for insertions and removals from either end of the container

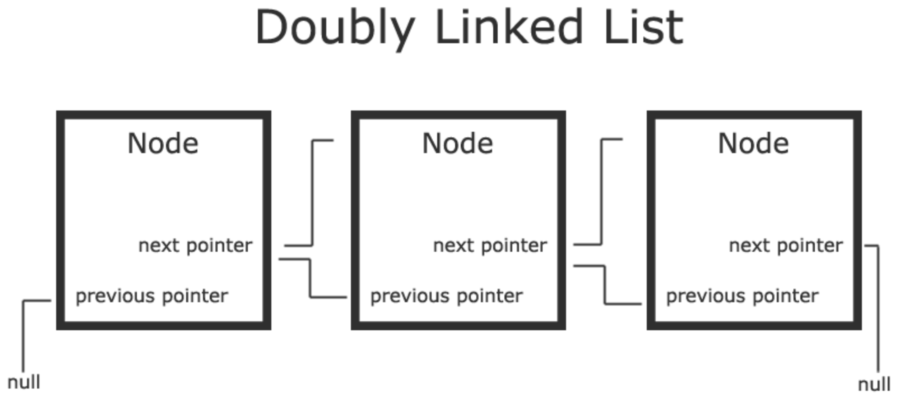


Double-Ended Link List

- Example

Doubly Linked Lists

- can move forward by use of a next pointer and backward by use of a previous pointer



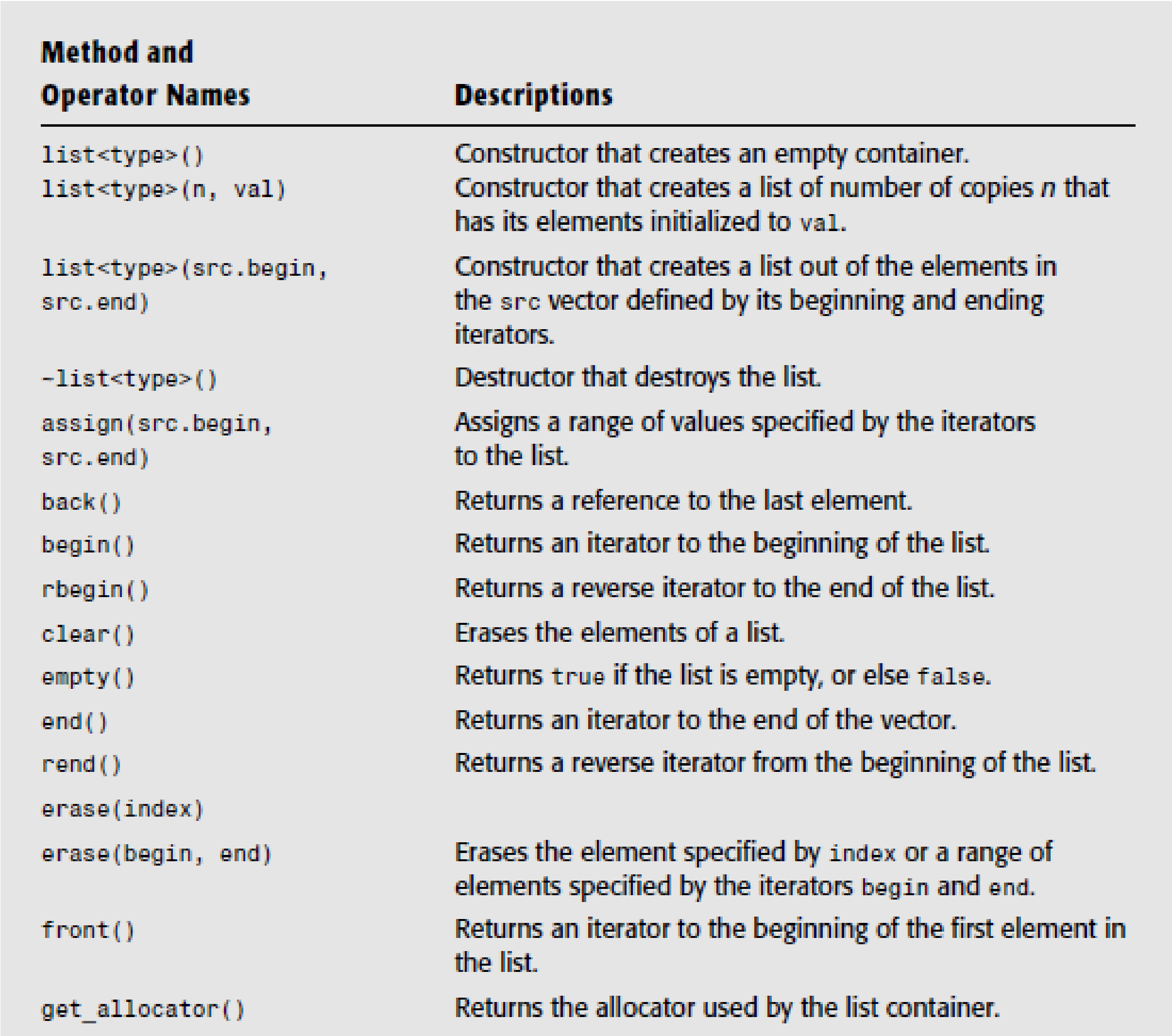
Doubly Linked Lists

- Example

STL Link List

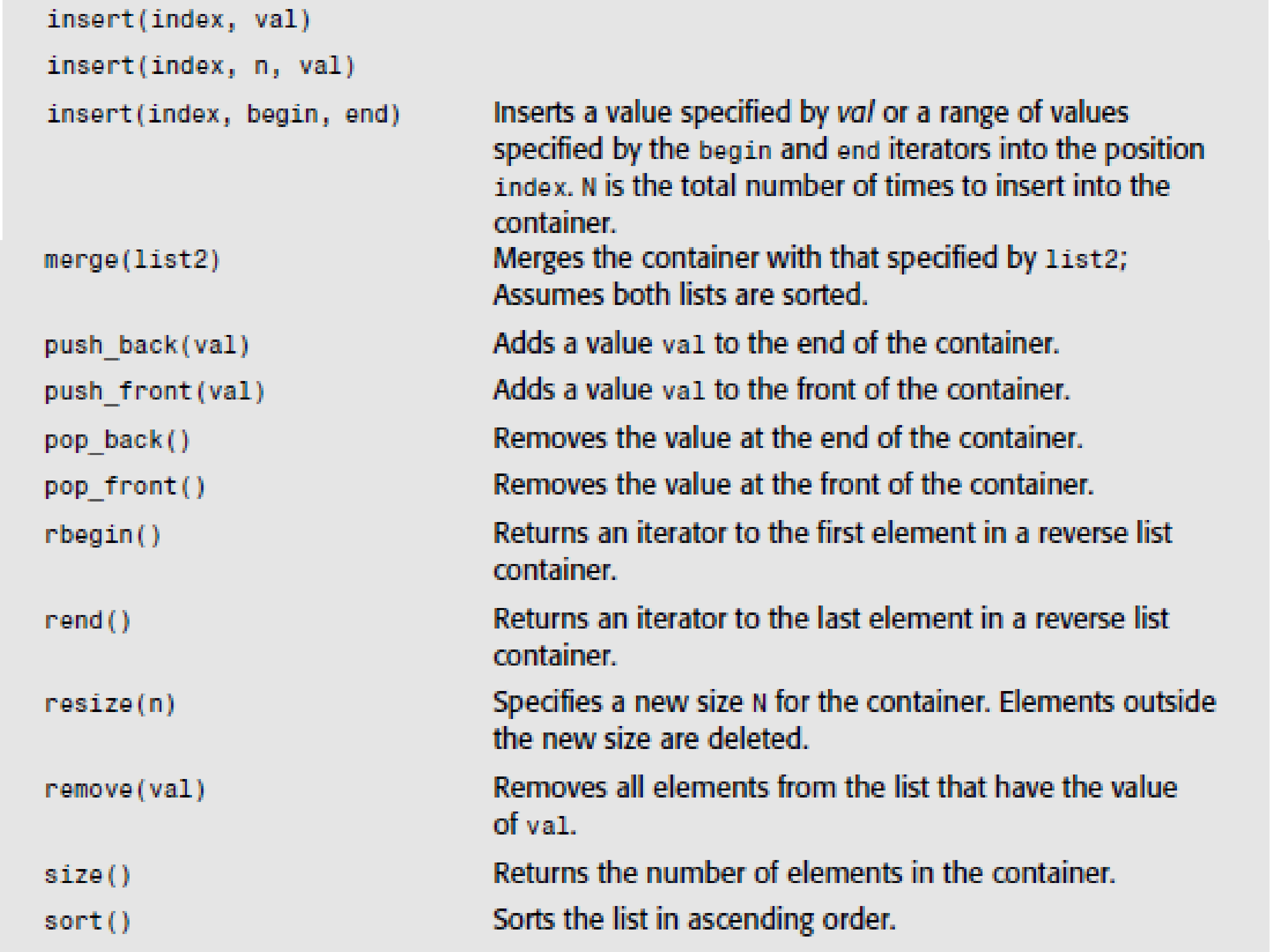
- a link list called the list container
- implemented as a doubly linked list that can push and pop elements from the front and the back of the container (double-ended)
- can be accessed by including the <list> header file

STL Link List



Method and Operator Names	Descriptions
`list<type>()`	Constructor that creates an empty container.
`list<type>(n, val)`	Constructor that creates a list of number of copies *n* that has its elements initialized to `val`.
`list<type>(src.begin, src.end)`	Constructor that creates a list out of the elements in the `src` vector defined by its beginning and ending iterators.
`-list<type>()`	Destructor that destroys the list.
`assign(src.begin, src.end)`	Assigns a range of values specified by the iterators to the list.
`back()`	Returns a reference to the last element.
`begin()`	Returns an iterator to the beginning of the list.
`rbegin()`	Returns a reverse iterator to the end of the list.
`clear()`	Erases the elements of a list.
`empty()`	Returns `true` if the list is empty, or else `false`.
`end()`	Returns an iterator to the end of the vector.
`rend()`	Returns a reverse iterator from the beginning of the list.
`erase(index)`	
`erase(begin, end)`	Erases the element specified by `index` or a range of elements specified by the iterators `begin` and `end`.
`front()`	Returns an iterator to the beginning of the first element in the list.
`get_allocator()`	Returns the allocator used by the list container.

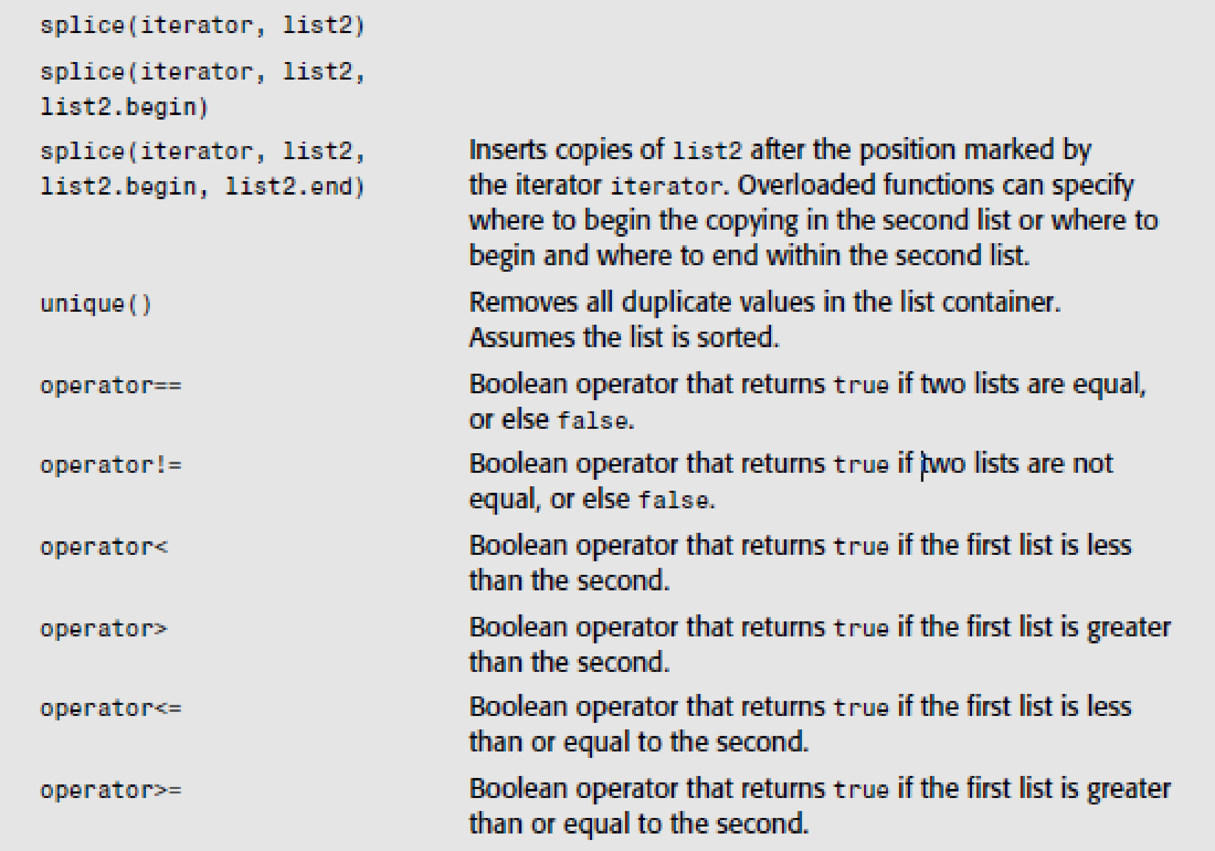
STL Link List



`insert(index, val)`
`insert(index, n, val)`
`insert(index, begin, end)`
	Inserts a value specified by *val* or a range of values specified by the `begin` and `end` iterators into the position `index`. `N` is the total number of times to insert into the container.

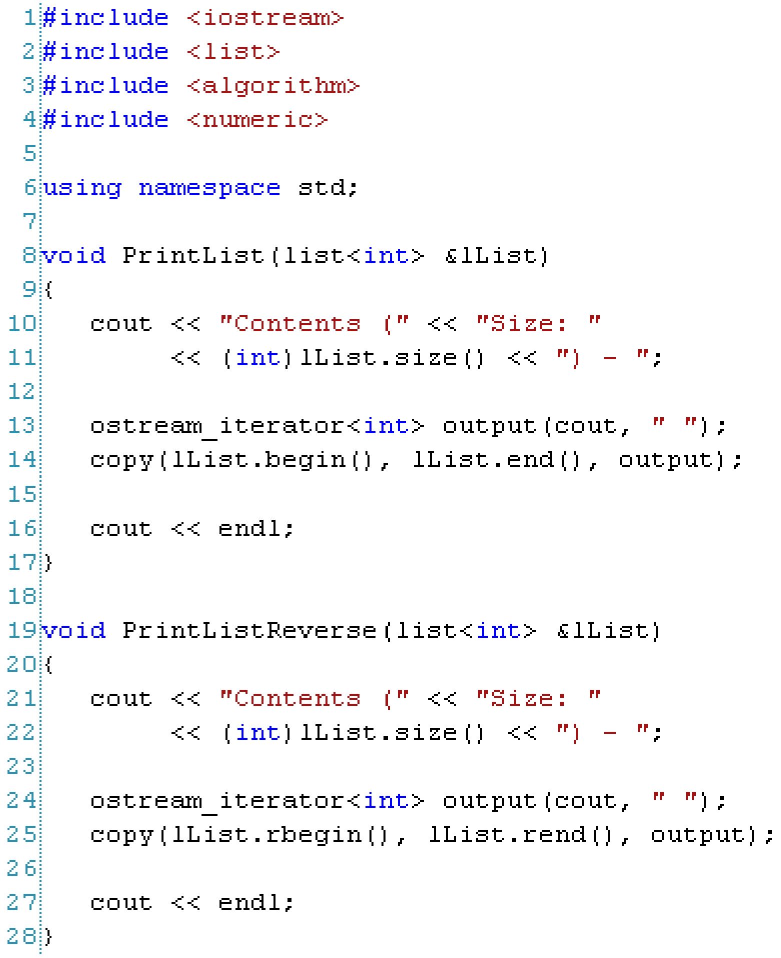
`merge(list2)`	Merges the container with that specified by `list2`; Assumes both lists are sorted.
`push_back(val)`	Adds a value `val` to the end of the container.
`push_front(val)`	Adds a value `val` to the front of the container.
`pop_back()`	Removes the value at the end of the container.
`pop_front()`	Removes the value at the front of the container.
`rbegin()`	Returns an iterator to the first element in a reverse list container.
`rend()`	Returns an iterator to the last element in a reverse list container.
`resize(n)`	Specifies a new size `N` for the container. Elements outside the new size are deleted.
`remove(val)`	Removes all elements from the list that have the value of `val`.
`size()`	Returns the number of elements in the container.
`sort()`	Sorts the list in ascending order.

STL Link List



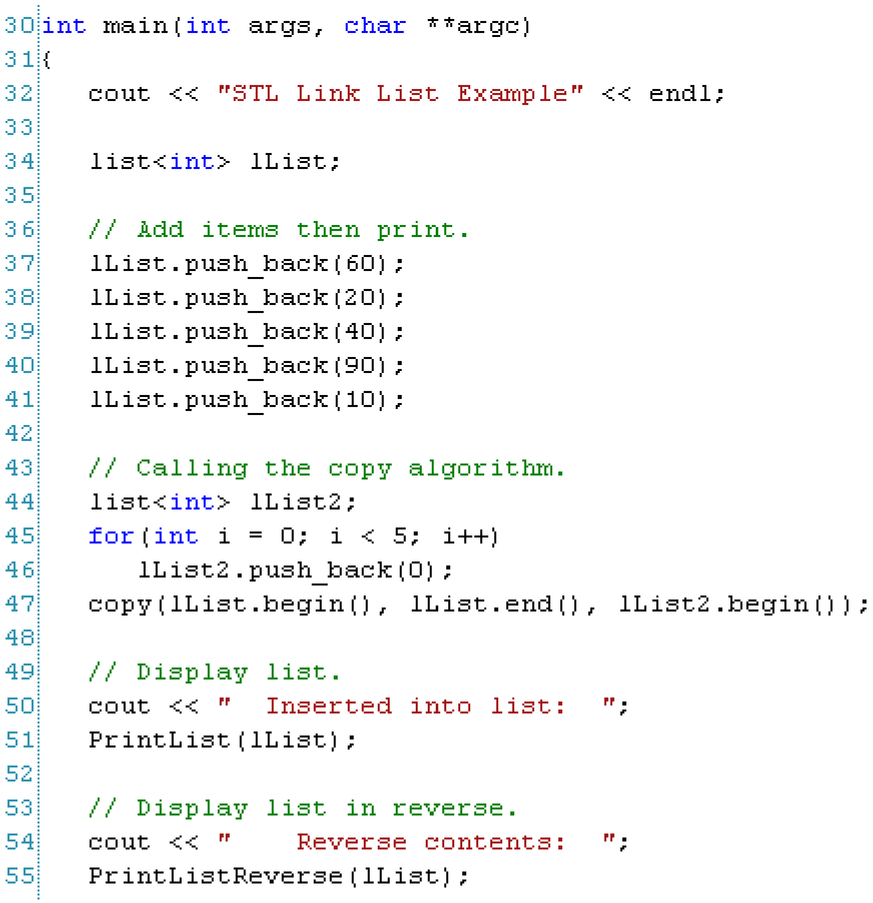
`splice(iterator, list2)`	
`splice(iterator, list2, list2.begin)`	
`splice(iterator, list2, list2.begin, list2.end)`	Inserts copies of `list2` after the position marked by the iterator `iterator`. Overloaded functions can specify where to begin the copying in the second list or where to begin and where to end within the second list.
`unique()`	Removes all duplicate values in the list container. Assumes the list is sorted.
`operator==`	Boolean operator that returns `true` if two lists are equal, or else `false`.
`operator!=`	Boolean operator that returns `true` if two lists are not equal, or else `false`.
`operator<`	Boolean operator that returns `true` if the first list is less than the second.
`operator>`	Boolean operator that returns `true` if the first list is greater than the second.
`operator<=`	Boolean operator that returns `true` if the first list is less than or equal to the second.
`operator>=`	Boolean operator that returns `true` if the first list is greater than or equal to the second.

STL Link List Example

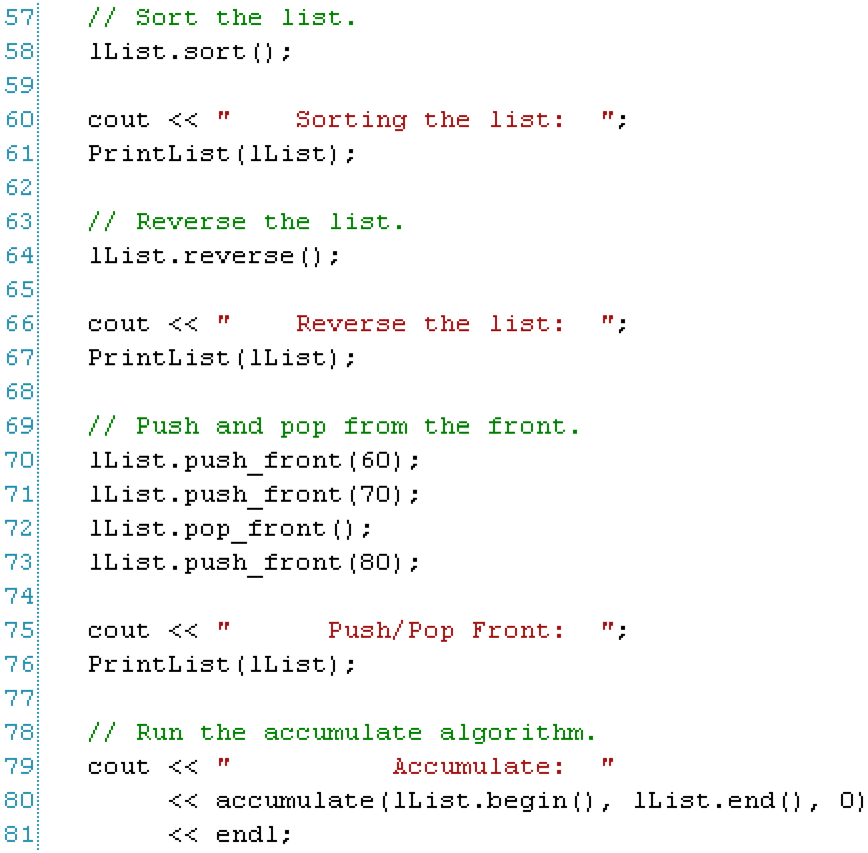


```cpp
#include <iostream>
#include <list>
#include <algorithm>
#include <numeric>

using namespace std;

void PrintList(list<int> &lList)
{
    cout << "Contents (" << "Size: "
         << (int)lList.size() << ") - ";

    ostream_iterator<int> output(cout, " ");
    copy(lList.begin(), lList.end(), output);

    cout << endl;
}

void PrintListReverse(list<int> &lList)
{
    cout << "Contents (" << "Size: "
         << (int)lList.size() << ") - ";

    ostream_iterator<int> output(cout, " ");
    copy(lList.rbegin(), lList.rend(), output);

    cout << endl;
}
```

STL Link List Example

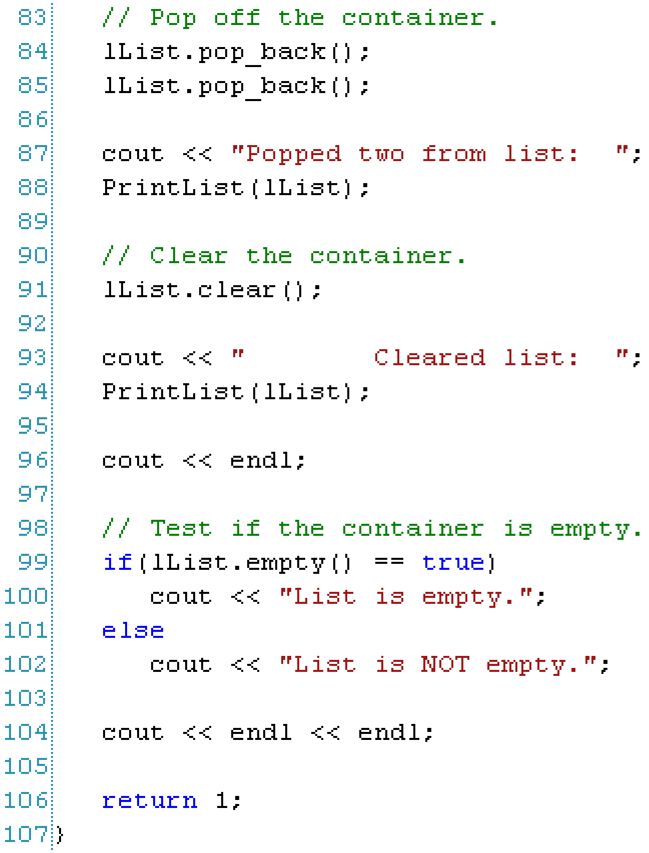


```
int main(int args, char **argc)
{
   cout << "STL Link List Example" << endl;

   list<int> lList;

   // Add items then print.
   lList.push_back(60);
   lList.push_back(20);
   lList.push_back(40);
   lList.push_back(90);
   lList.push_back(10);

   // Calling the copy algorithm.
   list<int> lList2;
   for(int i = 0; i < 5; i++)
      lList2.push_back(0);
   copy(lList.begin(), lList.end(), lList2.begin());

   // Display list.
   cout << "  Inserted into list:  ";
   PrintList(lList);

   // Display list in reverse.
   cout << "    Reverse contents:  ";
   PrintListReverse(lList);
```

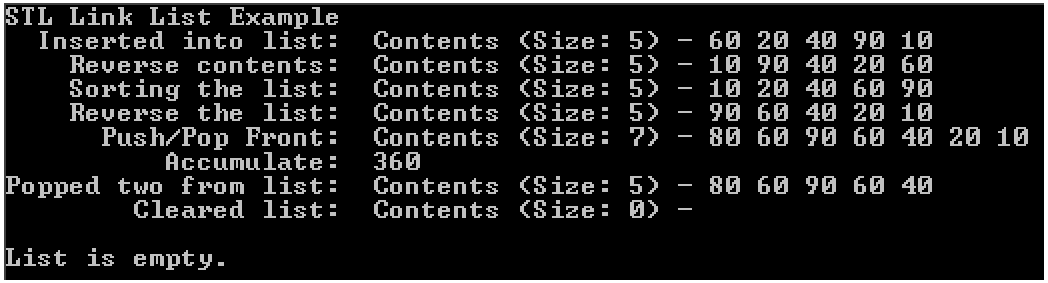


```
   // Sort the list.
   lList.sort();

   cout << "    Sorting the list:  ";
   PrintList(lList);

   // Reverse the list.
   lList.reverse();

   cout << "    Reverse the list:  ";
   PrintList(lList);

   // Push and pop from the front.
   lList.push_front(60);
   lList.push_front(70);
   lList.pop_front();
   lList.push_front(80);

   cout << "      Push/Pop Front:  ";
   PrintList(lList);

   // Run the accumulate algorithm.
   cout << "          Accumulate:  "
        << accumulate(lList.begin(), lList.end(), 0)
        << endl;
```

STL Link List Example



```
    // Pop off the container.
    lList.pop_back();
    lList.pop_back();

    cout << "Popped two from list:  ";
    PrintList(lList);

    // Clear the container.
    lList.clear();

    cout << "        Cleared list:  ";
    PrintList(lList);

    cout << endl;

    // Test if the container is empty.
    if(lList.empty() == true)
        cout << "List is empty.";
    else
        cout << "List is NOT empty.";

    cout << endl << endl;

    return 1;
}
```



```
STL Link List Example
  Inserted into list:  Contents (Size: 5) - 60 20 40 90 10
    Reverse contents:  Contents (Size: 5) - 10 90 40 20 60
    Sorting the list:  Contents (Size: 5) - 10 20 40 60 90
    Reverse the list:  Contents (Size: 5) - 90 60 40 20 10
      Push/Pop Front:  Contents (Size: 7) - 80 60 90 60 40 20 10
          Accumulate:  360
Popped two from list:  Contents (Size: 5) - 80 60 90 60 40
        Cleared list:  Contents (Size: 0) -

List is empty.
```

Link List

- fast insertions and deletions at the end and within the container
- can expand and shrink rapidly compared to arrays
- can be tighter in terms of memory than arrays, which can often allocate more memory than is needed
- slow to search
- do not have random access
- doubly linked lists have both forward and reverse iterators for the movement through the list
- link lists are made up of the data plus any pointers, which can result in very large lists (unlike arrays)