# Chp18 - The STL list container.

**Part 1 - Definitions**

**What is the std::list?**

- `The std::list` is an STL container implementing a doubly-linked list.

**What is a doubly-linked list?**

- A doubly-linked list is a linear data structure where each element (often called a **node**) in the sequence is connected to the element that comes before it and the element that comes after it (the neighbors).
- The list maintains references to the first node (called the *front* node) and the last node (*back* node).
- The bidirectional connection of std::list is the key characteristic that distinguishes it from a singly-linked list, which only has connections in one direction.

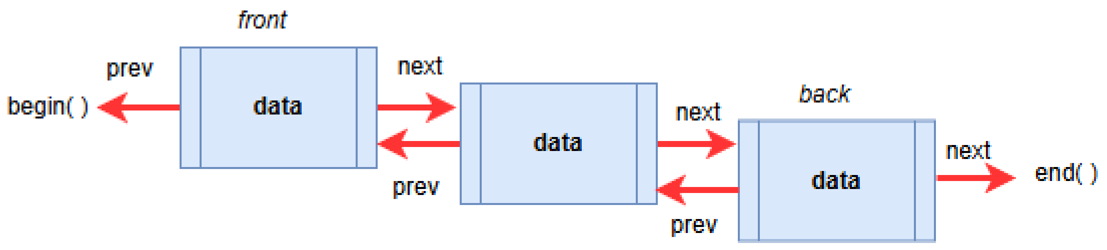


**Figure 1**. A doubly-linked list

**Architecture of the std::list**

**1. The Node:**

Each element in  the STL doubly-linked list is a template node containing three main components:

- **Data:** This holds the actual value or information that the node represents (e.g., a number, a string, an object).
- **Previous Pointer (prev):** This is a pointer (or reference) that stores the memory address of the **previous** node in the sequence. If the current node is the first node (the *front*), this pointer will typically be **NULL** or point to a special "dummy" node.
- **Next Pointer (next):** This is a pointer (or reference) that stores the memory address of the **next** node in the sequence. If the current node is the last node (the *back*), this pointer will typically be **NULL** (or a dummy node).

**Advantages of Doubly-Linked Lists:**

- **Efficient Insertion and Deletion:** Inserting or deleting nodes at any position in the list is efficient (O(1) time complexity *if* you already have a pointer to the location where you want to insert or delete). You only need to update the pointers of the neighboring nodes.
- **Bidirectional Traversal:** You can traverse the list in both forward and backward directions using the **prev** and **next** pointers, which can be useful for certain algorithms.
- **Easier Deletion (with a pointer to the node):** Deleting a node is simpler because you have direct access to both the previous and next nodes, so you can easily update their pointers.

**Disadvantages of Doubly-Linked Lists:**

- **More Memory Overhead:** Each node requires extra memory to store the prev pointer, compared to a singly-linked list.
- **Slightly More Complex Implementation:** The logic for insertion and deletion involves updating more pointers, making the implementation slightly more complex than a singly-linked list or a contiguous data structure like an array or vector.

**Part 2. Examples illustrating how the std::list works ( fourteen use cases are listed below)**

The following code examples use the **Person** struct depicted below

```
struct Person {
       string name;
       int    age;

       // Constructor
       Person(string n, int a) : name(n), age(a) {}

       // Comparison operator for sorting
       bool operator<(const Person& other) const {
              return age < other.age;  // Sort by age
       }

       friend ostream& operator<<(ostream& sout, const Person& p) {
              sout << p.name << ", " << p.age;
              return sout;
       }
};
```

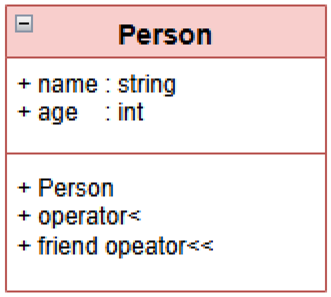


**Figure 2**. The Person class diagram

**Case 1. Defining and initializing a list.**

```
	// Case 1. Create a list of Simpsons characters
	list<Person> simpsons = {
		{"Homer Simpson", 39},
		{"Marge Simpson", 36},
		{"Bart Simpson", 10},
		{"Lisa Simpson", 8},
		{"Maggie Simpson", 1}
	};
```

This quick definition and initialization step is based on list-initializers. This approach is a better practice than instantiating temporary objects such as in

```
	list<Person> simpsons = {
		Person("Homer Simpson", 39),
	       // etc
	};
```

**Case2. Adding elements to the list**

```
	//Case 2. Adding elements
	simpsons.push_back({ "Abe Simpson", 83 });    //node added at the end of the list
	simpsons.push_front({ "Ned Flanders", 60 }); //node becomes the new head of the list
```

push_back and push_front demonstrate std::list's excellent efficiency for adding elements at either end, a key advantage over std::vector for front insertions.

**Case3. Removing elements from the list**

```
	//Case 3. Removing elements
	simpsons.pop_back();  // Remove last element
	simpsons.pop_front(); // Remove first element
```

pop_back and pop_front provide constant-time removal from both ends, mirroring the efficiency of their insertion counterparts in std::list.

**Case 4. Inserting at a given position (other than the front and back)**

```
	//Case 4. Insert a new person at the 'fourth' position
	auto it = simpsons.begin();
	advance(it, 4);  // Move iterator to the 'fourth' position 0..4..

	simpsons.insert(it, Person("Milhouse Van Houten", 10));
```

Inserting mid-list in std::list involves iterator manipulation but avoids the potentially expensive element shifting of std::vector.

**Case 5. Removing an element stored at a given position using erase**

```
	//Case 5. Removing an element using erase ('third' element - Lisa out)
	it = simpsons.begin();
	advance(it, 3);
	simpsons.erase(it);
```

Applying 'erase' at a location designated by a specific iterator highlights std::list's ability to efficiently remove elements from the middle without shifting subsequent elements.

**Case 6. Bulk removal using erase_if**

```
	//Case 6. Remove all persons aged below 10
	simpsons.remove_if([](const Person& p) { return p.age < 10; });
```

remove_if elegantly demonstrates how to efficiently eliminate multiple elements based on a condition, showcasing a powerful feature of std::list (though note: remove_if is a member function of std::list, not erase_if which is a generic algorithm).

**Case 7. Sorting using the collection's natural order**

```
	//Case 7. Sorting the list by age
	simpsons.sort();
```

This is an efficient strategy to sort the list either using the natural order of the collection (such as int, double, and strings do) or the pattern defined by the Person's overloaded 'operator<' which indicates how to compare two people.

**Case 8. Sorting using a custom order.**

```
	// Case 8. Sorting with a custom comparator(by name, then by age)
	simpsons.sort([](const Person& a, const Person& b) {
			if (a.name == b.name) {
			   return a.age < b.age;        // Sort by name first
			}
			   return a.name < b.name;      // use age to break any ties
			});
```

std::list's sort() accepts a custom comparator, enabling flexible sorting logic beyond the natural ordering of elements.

**Case 9. Reverse the list**

```
        //Case 9. Reversing the list
        simpsons.reverse();
```

reverse() efficiently flips the order of elements in a std::list by manipulating pointers, avoiding the need to copy or swap individual elements extensively.

**Case 10. Merge two lists**

```
        //Case 10. Create another list and merge them

        list<Person> springfield = {     // A new list of Springfield people
              {"Mr. Burns", 104},
              {"Barney Gumble", 45},
              {"Krusty the Clown", 50},
              {"Bart Simpson", 10},      // Note: intentionally duplicated
        };


        // Precondition - Sort before merging (note: age is used as natural order)
        simpsons.sort();
        springfield.sort();
        simpsons.merge(springfield);
```

merge() efficiently combines two *sorted* std::list objects into one, maintaining the sorted order without the overhead of repeatedly inserting individual elements.

**Case 11. Remove consecutive duplicate nodes**

```
        //Case 11. Remove consecutive duplicates based on the name and age
        simpsons.unique([](const Person& a, const Person& b) {
              // Custom unique: Remove duplicates based on name and age
              if (a.name == b.name) {
                    return a.age == b.age; // If names are the same, check ages
              }
              return false; // Otherwise, keep them
              });
```

unique() with a custom predicate efficiently eliminates *adjacent* duplicates based on a specific comparison, highlighting the importance of prior sorting for removing all duplicates.

**Case 12. Forward traversal using next()**

```
	//Case12. Traversing the list using next() iterator
	cout << "\nShowing first three entries using next() iterator:\n";
	auto itStart = simpsons.begin();				// Start iterator
	while (itStart != simpsons.end() &&
		distance(simpsons.begin(), itStart) < 3) {
		cout << *itStart << endl;				// Print the current element
		itStart = next(itStart);				// same as itStart++
	}
```

Using next() provides a functional-style alternative to advance iterators, offering a slightly different perspective on traversal compared to the traditional 'iterator++' approach.

**Case 13. Backward traversal using prev()**

```
	// Case 13. Traverse the list using prev() iterator
	cout << "\nShowing last three entries using prev() iterator:\n";
	int count = 0;
	auto itEnd = simpsons.end();			// End iterator (one past the last element)
	--itEnd;					// Move back to the previous element

	while (itEnd != simpsons.begin() && count < 3) {
		cout << *itEnd << endl;			// Print the current element
		count++;
		itEnd = prev(itEnd);			// Move to the previous element

	}
```

prev() elegantly facilitates backward iteration in std::list, directly leveraging its doubly-linked nature for efficient reverse traversal.

**Case 14. Splicing a list**

```
	//Case 14. Splicing: Moving first three elements to another list
	list<Person> vipList;
	vipList.splice(vipList.begin(), simpsons, simpsons.begin(), next(simpsons.begin(), 3));
	printList(vipList, "VIP List (First 3 from Simpsons)");
```

splice() offers a highly efficient way to transfer nodes between std::list objects without copying data, making it ideal for rearranging list segments quickly.

The splice() method for std::list comes in several overloaded forms. The one used here has the following signature

- Position: Where to insert in the destination (vipList.begin() means the start).
- Source: The list from which nodes will be taken (simpsons).
- First: Start of the range to move (simpsons.begin()).
- Last: End of the range to move (exclusive, next(begin(), 3) means up to the third element).

**Conclusion**

In summary, `std::list` is well-suited for applications that require frequent insertions and deletions, particularly in the middle of a sequence. However, due to the lack of efficient random access, it's important to use it with care in scenarios where fast element retrieval by index is needed.

## SOLUTION

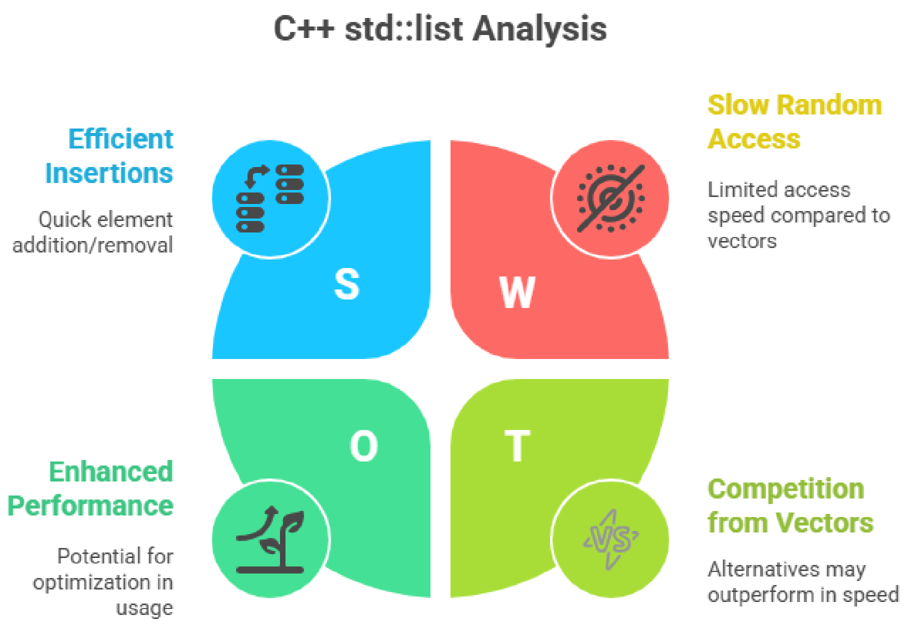
```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;


// Define the Person struct
struct Person {
	string name;
	int    age;

	// Constructor
	Person(string n, int a) : name(n), age(a) {}

	// Comparison operator for sorting
	bool operator<(const Person& other) const {
		return age < other.age;  // Sort by age
	}

	friend ostream& operator<<(ostream& sout, const Person& p) {
		//sout << " Person [Name:" << p.name << ", Age: " << p.age << "]";
		sout << p.name << ", " << p.age;
		return sout;
}
};

// Function to print a list
template <typename T>
void printList(const list<T>& people, const string& caption) {
	cout << caption << ":\n";
	for (const auto& p : people) {
		cout << "   " << p << endl;
	}
	cout << "-----------------------------\n";
}

int main() {
	// Case 1. Create a list of Simpsons characters
	list<Person> simpsons = {
		{"Homer Simpson", 39},
		{"Marge Simpson", 36},
		{"Bart Simpson", 10},
		{"Lisa Simpson", 8},
		{"Maggie Simpson", 1}
	};

	printList(simpsons, "Initial List");

	//Case 2. Adding elements
	simpsons.push_back({ "Abe Simpson", 83 });
	simpsons.push_front({ "Ned Flanders", 60 });
	printList(simpsons, "After push_front and push_back");
	cout << "Front element: " << simpsons.front() << endl; // Should be Ned Flanders
	cout << "Back element:  " << simpsons.back() << endl;   // Should be Abe Simpson
	cout << "List size:     " << simpsons.size() << endl; // Should be 7 now (5 original + 2
added)

	//Case 3. Removing elements
	simpsons.pop_back();  // Remove last element
	simpsons.pop_front(); // Remove first element
	printList(simpsons, "After pop_front and pop_back");
```

```cpp
//Cas 4. Insert a new person at the 'fourth' position
auto it = simpsons.begin();
advance(it, 4);  // Move iterator to 'fourth' position 0..4..
//simpsons.insert(it, { "Milhouse Van Houten", 10 });
simpsons.insert(it, Person("Milhouse Van Houten", 10)); // equivalent to above
printList(simpsons, "After inserting Milhouse at position 4");

//Case 5. Removing an element using erase ('third' element - Lisa out)
it = simpsons.begin();
advance(it, 3);
simpsons.erase(it);
printList(simpsons, "After erasing 'third' element");



//Case 6. Remove all persons aged below 10
simpsons.remove_if([](const Person& p) { return p.age < 10; });
printList(simpsons, "After removing people younger than 10");



//Case 7. Sorting the list by age
simpsons.sort();
printList(simpsons, "After sorting by age");

// Case 8. Sorting with a custom comparator(by name, then by age)
simpsons.sort([](const Person& a, const Person& b) {
      // Custom sort: Sort by name. Use age to break ties
      if (a.name == b.name) {
            return a.age < b.age; // Sort by name first, then age
      }
      return a.name < b.name; // Otherwise sort by age
      });
printList(simpsons, "After sorting by name");


//Case 9. Reversing the list
simpsons.reverse();
printList(simpsons, "After reversing the list");

//Case 10. Create another list and merge them
list<Person> springfield = {
      {"Mr. Burns", 104},
      {"Barney Gumble", 45},
      {"Krusty the Clown", 50},
      {"Bart Simpson", 10}, // Note: intentionally duplicated
};
printList(springfield, "Springfield Characters");

//Case 10. Sort (by age) before merging
simpsons.sort();
springfield.sort();
simpsons.merge(springfield);
printList(simpsons, "After merging Springfield list into Simpsons list");

//Case 11. Remove duplicates based on the name and age
simpsons.unique([](const Person& a, const Person& b) {
      // Custom unique: Remove duplicates based on name and age
      if (a.name == b.name) {
            return a.age == b.age; // If names are the same, check ages
      }
      return false; // Otherwise, keep them
      });
printList(simpsons, "After removing duplicates based on name and age");
```

```
    //Case12. Traversing the list using next() iterator
    cout << "\nShowing first three entries using next() iterator:\n";
    auto itStart = simpsons.begin();       // Start iterator
    while (itStart != simpsons.end() &&
            distance(simpsons.begin(), itStart) < 3) {
        cout << "  " << *itStart << endl;      // Print the current element
        ++itStart;                                           // Move to the next element
    }
    cout << "----------------------------\n";

    // Case 13. Traverse the list using prev() iterator
    cout << "\nShowing last three entries using prev() iterator:\n";
    int count = 0;
    auto itEnd = simpsons.end(); // End iterator (one past the last element)
    while (itEnd != simpsons.begin() && count < 3) {
        --itEnd; // Move back to the previous element
        cout << "  " << *itEnd << endl; // Print the current element
        ++count;
    }
    cout << "----------------------------\n";


    //Case 14. Splicing: Moving first three elements to another list
    list<Person> vipList;
    vipList.splice(vipList.begin(), simpsons, simpsons.begin(), next(simpsons.begin(), 3));
    printList(vipList, "VIP List (First 3 from Simpsons)");
    printList(simpsons, "Simpsons List after Splicing");

    cout << "\nAll done!\n";
}
```

**SWOT Analysis**

| | |
|---|---|
| Strengths: | Quick element addition/removal at any place in the list. |
| Weakness: | Limited random-access support compared to vectors |
| Opportunities: | Potential for optimization in usage. |
| Threats: | Alternative sequence containers may outperform in speed. You **can't use `[]` or `at()`** like in vectors. |


C++ std::list Analysis
Efficient Insertions
Quick element addition/removal
S
Slow Random Access
Limited access speed compared to vectors
W
Enhanced Performance
Potential for optimization in usage
O
Competition from Vectors
Alternatives may outperform in speed
T
VS