friend function:

Concept:

A **friend function** is a non-member function that has access to private and protected members of a class.

Code Example:

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

class Demo {
private:
    int value;

public:
    Demo(int v) : value(v) {}

    // Declare friend function
    friend void showValue(Demo d);
};

// Friend function definition
void showValue(Demo d) {
    cout << "Value: " << d.value << endl;
}

int main() {
    Demo obj(10);
    showValue(obj); // Friend function accessing private member
    return 0;
}
```

Output:

```
Value: 10
```

Explanation:

- showValue(Demo d) is **not a member** of Demo but is declared as a **friend**.
- This allows it to access the private member value of Demo.
- The function prints the value of value when called in main().

In C++, **streams** are used to perform input and output (I/O) operations. A stream is essentially a flow of data, which can either be **input stream** (data flows into the program) or **output stream** (data flows out of the program).

Types of Streams in C++

C++ provides various types of streams for handling different kinds of I/O operations:

1. Input Streams (istream)

- Used to read data from input sources like the keyboard or files.
- Example: cin (standard input)

2. Output Streams (ostream)

- Used to write data to output destinations like the console or files.
- Example: cout (standard output)

In C++, the **stream classes** (like iostream) provide two ways to handle input and output:

1. **Formatted I/O** (default) – Uses formatting rules to properly display data.
2. **Unformatted I/O** – Reads or writes raw sequences of characters without formatting.

---

1. Formatted I/O

Uses stream manipulators (std::setw, std::setprecision, std::fixed, etc.) to control output formatting.

Example:

```
#include <iostream>
#include <iomanip>  // Required for formatting
using namespace std;

int main() {
   double num = 123.456;

   // Formatted output
   cout << "Formatted Output:\n";
   cout << "Fixed: " << fixed << setprecision(2) << num << endl;
   cout << "Scientific: " << scientific << num << endl;
   cout << "Width 10: " << setw(10) << num << endl;

   return 0;
}
```

Output:

```
Formatted Output:
Fixed: 123.46
Scientific: 1.234560e+02
Width 10:     123.46
```

---

2. Unformatted I/O

Works with raw character sequences using functions like get(), put(), read(), write(), etc.

Example:

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

int main() {
    char ch;

    cout << "Enter a character: ";
    ch = cin.get();  // Unformatted input
    cout.put(ch);    // Unformatted output

    return 0;
}
```

Output:

```
Enter a character: A
A
```

Here, cin.get() reads a single character, including spaces, and cout.put() outputs it.

Comparison Table:

Feature	**Formatted I/O**	**Unformatted I/O**
Functions Used	<<, >>, setw(), setprecision()	get(), put(), read(), write()
Handles Formatting?	Yes	No
Performance	Slower (due to formatting overhead)	Faster
Use Case	When formatting is needed (e.g., tables, precision)	When raw character handling is required (e.g., file I/O, binary data)

STL (Standard Template Library) in C++

The **Standard Template Library (STL)** is a powerful set of C++ template classes that provide common **data structures** and **algorithms**. It allows developers to use efficient implementations of frequently used operations such as sorting, searching, and managing collections.

Components of STL

STL is divided into **three main components**:

1. Containers

- Containers are used to store collections of objects.
- They are implemented as **templates**, allowing flexibility with different data types.
- Containers are categorized into:
  - **Sequence Containers**: Store data in linear order (e.g., vector, list, deque).
  - **Associative Containers**: Store data in sorted order (e.g., set, map).
  - **Unordered Containers**: Store data without any specific order (e.g., unordered_set, unordered_map).

Container Type	**Examples**
Sequence Containers	vector, list, deque
Associative Containers	set, map, multiset, multimap
Unordered Containers	unordered_set, unordered_map, unordered_multiset, unordered_multimap

---

2. Algorithms

- STL provides many built-in **algorithms** that work with containers.
- Examples: Sorting, searching, modifying, counting, and manipulating collections.

Algorithm Category	**Examples**
Sorting	sort(), stable_sort()
Searching	find(), binary_search()
Modifying	replace(), fill(), copy()
Counting	count(), count_if()
Numeric Operations	accumulate(), inner_product()

---

3. Iterators

- Iterators act like **pointers** to traverse containers.
- Types of iterators:
  - **Input Iterator** – Read values sequentially (istream_iterator).
  - **Output Iterator** – Write values sequentially (ostream_iterator).
  - **Forward Iterator** – Can only move forward (forward_list).
  - **Bidirectional Iterator** – Can move forward and backward (list, set).
  - **Random Access Iterator** – Can access any element (vector, deque).

STL Components in This Demo:

1. **vector** → Dynamic array.
2. **list** → Doubly linked list.
3. **set** → Stores unique elements in sorted order.
4. **map** → Key-value pair storage.
5. **algorithm** → Using sort(), find(), and count().

C++ STL Demo Code

```
#include <iostream>
#include <vector>
#include <list>
#include <set>
#include <map>
#include <algorithm>

using namespace std;

int main() {
    // 1. Vector Demo (Dynamic Array)
    vector<int> numbers = {10, 20, 30, 40, 50};
    numbers.push_back(60);  // Add element at the end
    cout << "Vector Elements: ";
    for (int num : numbers) cout << num << " ";
    cout << endl;

    // 2. List Demo (Doubly Linked List)
    list<string> names = {"Alice", "Bob", "Charlie"};
    names.push_front("Zara");  // Insert at the beginning
    cout << "List Elements: ";
    for (const string& name : names) cout << name << " ";
    cout << endl;

    // 3. Set Demo (Unique Sorted Elements)
    set<int> uniqueNumbers = {50, 10, 30, 20, 40};
    uniqueNumbers.insert(20);  // Duplicate won't be added
    cout << "Set Elements: ";
    for (int num : uniqueNumbers) cout << num << " ";
    cout << endl;

    // 4. Map Demo (Key-Value Pair)
    map<int, string> studentMap;
    studentMap[101] = "Alice";
    studentMap[102] = "Bob";
    studentMap[103] = "Charlie";
    cout << "Map Elements:\n";
    for (auto& [id, name] : studentMap)
        cout << "ID: " << id << ", Name: " << name << endl;
    // 5. Algorithm Demo
    vector<int> nums = {5, 3, 8, 1, 9};
    sort(nums.begin(), nums.end()); // Sort the vector
    cout << "Sorted Vector: ";
    for (int num : nums) cout << num << " ";
    cout << endl;

    // Find an element
    auto it = find(nums.begin(), nums.end(), 8);
    if (it != nums.end())
        cout << "Element 8 found at index: " << distance(nums.begin(), it) << endl;
    else
        cout << "Element 8 not found." << endl;

    return 0;
}
```

Output

```
Vector Elements: 10 20 30 40 50 60
List Elements: Zara Alice Bob Charlie
Set Elements: 10 20 30 40 50
Map Elements:
ID: 101, Name: Alice
ID: 102, Name: Bob
ID: 103, Name: Charlie
Sorted Vector: 1 3 5 8 9
Element 8 found at index: 3
```