QUESTION :01

Function and Class Templates:

1.Function Templates:

Function templates in C++ allow you to write generic functions that can operate on different data types without having to write separate code for each type. They are defined using the template keyword, followed by a template parameter list, and can be used with various data types.

Example:

```
template <typename T>

T add(T a, T b) {

   return a + b;

}
```

2.Class Templates vs Function Templates:

- Function templates are used to define generic functions.
- Class templates are used to define generic classes, where the types of one or more members can be specified at compile time.

3. Advantages of Templates in C++:

- Code Reusability: Templates allow you to write generic code that can be reused with different data types.
- Type Safety: The compiler ensures type safety during template instantiation.
- Flexibility: Templates provide flexibility in working with different data types without sacrificing performance.

4. Template Parameters with Default Values:

- Yes, template parameters can have default values. Default template arguments are specified in the template declaration.

Example:

```
template <typename T = int>

class MyClass {

   // class definition

};
```

Static Members:

1. static Member in a Class:

A static member in a class is a member that belongs to the class rather than an instance of the class. It is shared among all instances of the class.

2. Difference from Instance Member:

- Instance members are specific to each object, while static members are shared by all objects of the class.
- Static members are accessed using the class name, not through an instance.

3. Static Function in C++:

A static function in C++ is a member function that belongs to the class rather than an instance. It can be called using the class name, without creating an object.

4. Purpose of Static Variable:

- Static variables are shared among all instances of a class.
- They retain their values between different calls to the function or instances of the class.

Example:

```
class Example {
public:
   static int count; // static variable declaration
};

int Example::count = 0; // static variable definition
```

Friend Functions and Classes:

1.Friend Function in C++:

A friend function in C++ is a function that is not a member of a class but is granted access to its private and protected members.

2. Difference from Member Function:

- Friend functions are not members of the class but can access its private members.
- Member functions are part of the class and have direct access to private member.

3. Access to Private Members:

Yes, a friend function can access the private members of a class.

4. Friend Class:

A friend class is a class that is declared as a friend of another class. This allows the friend class to access the private and protected members of the class it is a friend of.

Example:

```
class My Class {
   friend class Friend Class; // Friend Class is a friend of My Class
```

```
    int private Member;

};
```

STL:

1.Standard Template Library (STL) in C++:

The Standard Template Library (STL) is a set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures.

2. Difference between Vector and List:

- Std: : vector is a dynamic array with contiguous memory, providing fast random access.
- std::list is a doubly-linked list, allowing for efficient insertions and removals at both ends.

3. Advantages of Using Vectors over Arrays:

- Dynamic Size: Vectors can dynamically resize, whereas array size is fixed.
- Automatic Memory Management: Vectors handle memory allocation and deallocation automatically.
- Range Checking: Vectors provide bounds checking on element access.

4. Dynamically Resize a Vector:

To dynamically resize a vector, you can use the resize or pushback method.

Example:

```
#include <vector>


int main() {

    std::vector<int> myVector;

    myVector.resize(10); // Resizes vector to size 10


    // OR


    MyVector.push back(42); // Adds element to the end, dynamically resizing if necessary

}
```

QUESTION :02

TASK :01

1.STL (Vectors):

SOLUTION:

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

struct Product {
   int id;
   std::string name;
   // Add other product details as needed
};

std::vector<Product> inventory;

void add Product (int id, const std :: string& name) {
   Product new Product {id ,name };
   Inventory. pushback (new Product);
}

void remove Product (int id) {
   auto it = std:: remove_ if(inventory .begin(), inventory  .end(),
                         [id] (const Product& p) { return p.id == id; });
   inventory. Erase (it, inventory. end());
}

int main() {
   // Example usage
   Add Product (1, "Laptop");
   Add Product (2, "Smartphone");

   Std ::cout << "Initial Inventory:\n";
```

```
    for (const auto& product : inventory) {
        std::cout << "ID: " << product.id << ", Name: " << product.name << "\n";
    }

    Remove Product(1);

    std::cout << "\n Inventory after removing product with ID 1:\n";
    for (const auto& product : inventory) {
        std:: cout << "ID: " << product.id << ", Name: " << product.name << "\n";
    }

    return 0;
}
```

TASK:02

STL (Algorithms):

SOLUTION:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <random>

// Function to initialize a vector with descending order
std::vector<int> initialize Descending Vector(int size) {
    std::vector<int> descending Vector;
    for (int i = size; i > 0; --i) {
        descending Vector. push_ back(i);
    }
    return descending Vector;
}
```

```
int main() {
    const int vector Size = 100000;
    std::vector<int> data Vector = initialize Descending Vector(vector Size);

    // Start the clock
    auto start Time Bubble Sort = std::chrono::high_ resolution_ clock::now();

    // Bubble Sort
    // ... Implement Bubble Sort algorithm here

    // Stop the clock
    auto end Time Bubble Sort = std::chrono::high_ resolution_ clock::now();
    auto duration Bubble Sort =
std::chrono::duration_cast<std::chrono::microseconds>(endTimeBubbleSort –
start Time Bubble Sort);
    std::cout << "Bubble Sort Execution Time: " << duration Bubble Sort. count()
<< " microseconds\n";

    // Start the clock
    auto start Time STL Sort = std::chrono::high_ resolution_ clock::now();

    // STL Sort
    std::sort(data Vector. begin(), data Vector. end());

    // Stop the clock
    auto end Time STL Sort = std::chrono::high_ resolution_ clock::now();
    auto duration STL Sort =
std::chrono::duration_cast<std::chrono::microseconds>(endTimeSTLSort –
start Time STL Sort);
    std::cout << "STL Sort Execution Time: " << duration STL Sort .count() << "
microseconds\n";
```

```
    // Print first 10 and last 10 integers
    std::cout << "First 10 Integers: ";
    for (int i = 0; i < 10; ++i) {
        std::cout << data Vector[i] << " ";
    }
    std::cout << "\n Last 10 Integers: ";
    for (int i = vector Size - 10; i < vector Size; ++i) {
        std::cout << data Vector[i] << " ";
    }

    return 0;
}
```