DAY – 8 Assignments and LAB's Task

Assignment -1

Problem Statement:

1. Find Minimum Element:

Problem: Write a generic function template named findMinimum in C++ that takes an array any data type T and its size n as arguments. The function should return the minimum element present in the array.

2. Swap Elements:

Problem: Write a function template swap that takes two pointers to variables of any data type T and swaps their values.

Constraints: The function should only modify the values pointed to by the arguments, not the arguments themselves (pass by reference).

3. Find Maximum Element:

Problem: Similar to findMinimum, create a function template findMaximum that returns the maximum element in an array of any data type T.

```
#include <iostream>

#include <string>

using namespace std;
```

1. Code for Min Element using a Generic Template

```
template <typename T> T findMin(T* array, int n) { // Declares a template with a type parameter 'T'

    T min = array[0];

    for (int i = 1; i < n; i++) {

        if (array[i] < min) {
```

```
            min = array[i];
        }
    }
    return min;
}
int main() {
    int arr[] = {3, 1, 4, 9, 2};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Size of array is : "<<n<<endl;   // or use cout<<"\n"; for new line
    cout << "Minimum element is : "<<findMin(arr, n) <<endl;
    return 0;
}
```

2. Code for swapping using a Swap Template

```
template <typename T>
void swap(T* a, T* b) {      // void implies that thr funct. doesn't returns any value.
    T temp = *a;
    *a = *b;
    *b = temp;
}


int main() {
    int x = 10;
    int y = 20;
    cout << "Before swap: x = " << x << ", y = " << y <<endl;
    swap(&x, &y);
    cout << "After swap: x = " << x << ", y = " << y << endl;
```

```
    return 0;
}
```

2. (i) Code for swapping using a Swap Template (Using References)

```
template <typename T>
void swap(T& a, T& b) {      // void implies that thr funct. doesn't returns any value.
    T temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 10;
    int y = 20;
    cout << "Before swap: x = " << x << ", y = " << y <<endl;
    swap(x, y);
    cout << "After swap: x = " << x << ", y = " << y << endl;
    return 0;
}
```

Code - 3 Code for Max Element using a Max Template

```
template <typename T>
T findMax(T* array, int n) {    // function- findMaximum
    T max = array[0];
    for (int i = 1; i < n; ++i) {
        if (array[i] > max) {      // current elem. is max than max elem.
```

```
            max = array[i];
        }
    }
    return max;
}

int main() {
    int arr[] = {3, 1, 4, 9, 2};
    int n = sizeof(arr) / sizeof(arr[0]); // No. of elements in the array.
    cout << "Size of array is : "<<n<<endl;
    cout << "Maximum element is : " << findMax(arr, n) << endl;
    return 0;
}
```

Code 1. Basic Code on Use of Function Template .

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
int main() {
    int x = 4, y = 16;
    double a = 9.5, b = 10.5;

    cout << "Sum of Integer Values : " << add(x, y) << endl;
    cout << "Sum of Double Values : " << add(a, b) << endl;
```

```
    return 0;
}
```

Code 2 : Restriction on Generic Template

Generic template can't be used because here both the functions have different functionalities .

```
void fun(double a){
    cout<<" The value of a is : "<<a<<endl;
}

void fun(int b){
    if(b%2==0)
    {
        cout<<" Number is even";
    }
    else
    {
        cout<<" Number is odd";
    }
}
int main(){
    fun(4.6);
    fun(6);
    return 0;
}
```

Code 3 : Class Template Example.

```
template<class T>
class A
{
  public:
  T num1 = 5;
  T num2 = 6;
  void add()
  {
      cout<<"Addtion of num1 and num2:"<<num1+num2<<endl;
  }
};
int main()
{
    A<int>d;
    d.add();
    return 0;
}
```

Code 4 : Class Template with Multiple parameters.

```
template<class T1,class T2>
class A {
    T1 a;
    T2 b;
    public:
    A (T1 x,T2 y)
```

```
    {
        a = x;
        b = y;
    }
    void display()
    {
        cout<<"Values of a and b are "<<a<<" and "<<b<<endl;
    }

};

int main()
{
    A<int,float>d(5,6.5);  // 'd' is the object of class A
    d.display();
    return 0;
}
```

Assignment - 2

Problem : Design a generic data processing library using class and function templates in C++. This library should be able to handle various data types (e.g., integers, floats, strings) without code duplication.

Requirements:

Create a class template named DataContainer that can hold elements of any data type specified during instantiation. Implement member functions for DataContainer:

DataContainer(size_t size): Constructor to initialize the container with a specific size.T& operator[](size_t index): Overloaded subscript operator to access elements.void printAll(): Prints all elements of the container. Create a function template named swap that takes two

DataContainer objects as arguments and swaps their elements. Ensure proper memory management using appropriate constructors and destructors.

```
#include <vector>
//Class template DataContainer
template <typename T>
class DataContainer {
private:
    std::vector<T> elements;

public:
    // Constructor to initialize with a specific size
    DataContainer(size_t size) : elements(size) {}

    // Overloaded subscript operator to access elements
    T& operator[](size_t index) {
        return elements[index];
    }
    // Function to print all elements of the container
    void printAll() {
        std::cout << "Elements:";
        for (const auto& elem : elements) {
            std::cout << " " << elem;
        }
        std::cout << std::endl;
   }
};
```

```
// Function template swap for DataContainer objects
template <typename T>
void swap(DataContainer<T>& dc1, DataContainer<T>& dc2) {
    swap(dc1, dc2);
}

int main() {
    // Example usage
    DataContainer<int> dc1(5);
    DataContainer<int> dc2(5);

    // Initialize elements
    for (size_t i = 0; i < 5; ++i) {
        dc1[i] = i + 1;
        dc2[i] = i * 2;
    }
    // Print initial elements
    std::cout << "Before swapping:" << std::endl;
    dc1.printAll();
    dc2.printAll();
    // Swap containers
    swap(dc1, dc2);

    // Print swapped elements
    std::cout << "After swapping:" << std::endl;
    dc1.printAll();
```

```
    dc2.printAll();

    return 0;

}
```

Assignment – 3

Implement the DataContainer class template:

Define the template parameter to specify the data type. Use an array or a vector internally to store the elements. Implement the constructor, subscript operator, and printAll function as described in the requirements. Implement the swap function template: Take two DataContainer objects as arguments. Use a loop or recursion to iterate over corresponding elements and swap their values. Consider potential edge cases (e.g., containers of different sizes).

Write a main function to demonstrate the library: Create instances of DataContainer for different data types (e.g., int, float, string). Populate the containers with sample data. Call printAll on each container to verify its contents. Use the swap function to swap elements between containers of the same type. Print the containers again to confirm the swap.

Enhance the DataContainer class: Add member functions for: size(): Returns the current size of the container. push_back(const T& value): Appends an element to the back of the container (dynamically resize if necessary). Modify the constructor to accept an optional initial size (default to 0).

```
#include <vector>

#include <stdexcept>

#include <algorithm>  // For std::swap


// Class template DataContainer

 template <typename T>

class DataContainer {

private:

    std::vector<T> elements;


public:
```

```
// Constructor with optional initial size (default to 0)
DataContainer(size_t size = 0) : elements(size) {}

// Subscript operator to access elements
T& operator[](size_t index) {
    if (index >= elements.size()) {
        throw std::out_of_range("Index out of range");
    }
    return elements[index];
}

// Function to print all elements of the container
void printAll() const {
    std::cout << "Elements:";
    for (const auto& elem : elements) {
        std::cout << " " << elem;
    }
    std::cout << std::endl;
}

// Function to return current size of the container
size_t size() const {
    return elements.size();
}

// Function to append an element to the back of the container
```

```
    void push_back(const T& value) {
        elements.push_back(value);
    }
};

// Function template swap for DataContainer objects
template <typename T>
void swap(DataContainer<T>& dc1, DataContainer<T>& dc2) {
    if (dc1.size() != dc2.size()) {
        throw std::invalid_argument("Containers must have the same size to swap elements");
    }
    for (size_t i = 0; i < dc1.size(); ++i) {
        std::swap(dc1[i], dc2[i]);
    }
}

// Main function to demonstrate the DataContainer class template
int main() {
    // Creating instances of DataContainer for different data types
    DataContainer<int> dc_int;
    DataContainer<float> dc_float;
    DataContainer<std::string> dc_string;

    // Populating the containers with sample data
    for (int i = 1; i <= 5; ++i) {
        dc_int.push_back(i);
```

```
    dc_float.push_back(i * 1.1f);
    dc_string.push_back("Element_" + std::to_string(i));
}

// Printing all elements before swapping
std::cout << "Data in containers before swapping:" << std::endl;
dc_int.printAll();
dc_float.printAll();
dc_string.printAll();

// Swapping elements between containers of the same type
DataContainer<int> dc_int2;
for (int i = 10; i < 15; ++i) {
    dc_int2.push_back(i);
}

std::cout << "Before swapping dc_int and dc_int2:" << std::endl;
dc_int.printAll();
dc_int2.printAll();

try {
    swap(dc_int, dc_int2);
    std::cout << "After swapping dc_int and dc_int2:" << std::endl;
    dc_int.printAll();
    dc_int2.printAll();
} catch (const std::exception& e) {
```

```
        std::cerr << e.what() << std::endl;
    }
    // Demonstrating additional functionalities
    std::cout << "Size of dc_string: " << dc_string.size() << std::endl;
    dc_string.push_back("New Element");
    std::cout << "After adding a new element to dc_string:" << std::endl;
    dc_string.printAll();

    return 0;
}
```

Assignment – 4

Problem - Implement a class template for linked lists or binary search trees, leveraging the DataContainer class. Create function templates for generic sorting algorithms (e.g., bubble sort, selection sort).

```
 template <typename T>
class LinkedList : public DataContainer<T> {
private:
    struct Node {
        T data;
        Node* next;
    };

    Node* head;

public:
    LinkedList() : head(nullptr) {}
```

```
    // Insert a new element at the end of the list
    void insert(const T& value) {
        Node* newNode = new Node{value, nullptr};
        if (head == nullptr) {
            head = newNode;
        } else {
            Node* current = head;
            while (current->next != nullptr) {
                current = current->next;
            }
            current->next = newNode;
        }
    }

    // Print the linked list
    void printList() const {
        Node* current = head;
        while (current != nullptr) {
            std::cout << current->data << " ";
            current = current->next;
        }
        std::cout << std::endl;
    }
};

// Function template for bubble sort
```

```
template <typename T>
void bubbleSort(DataContainer<T>& container) {
    int n = container.size();
    for (int i = 0; i < n - 1; ++i) {
        for (int j = 0; j < n - i - 1; ++j) {
            if (container[j] > container[j + 1]) {
                std::swap(container[j], container[j + 1]);
            }
        }
    }
}

// Function template for selection sort
template <typename T>
void selectionSort(DataContainer<T>& container) {
    int n = container.size();
    for (int i = 0; i < n - 1; ++i) {
        int minIndex = i;
        for (int j = i + 1; j < n; ++j) {
            if (container[j] < container[minIndex]) {
                minIndex = j;
            }
        }
        std::swap(container[i], container[minIndex]);
    }
}
```

```
int main() {
    LinkedList<int> list;
    list.insert(5);
    list.insert(2);
    list.insert(8);
    list.insert(3);
    list.insert(1);

    std::cout << "Linked List: ";
    list.printList();

    DataContainer<int> container(5);
    container[0] = 5;
    container[1] = 2;
    container[2] = 8;
    container[3] = 3;
    container[4] = 1;

    std::cout << "DataContainer: ";
    container.printAll();

    bubbleSort(container);
    std::cout << "Sorted DataContainer (Bubble Sort): ";
    container.printAll();
```

```
    selectionSort(container);
    std::cout << "Sorted DataContainer (Selection Sort): " ;
    container.printAll();

    return 0;
}
```

Code – 5 Program on concept of Smart Pointers.

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

// A generic smart pointer class
template <  class T>
class Smartpointer {
    T *p; // Actual pointer
    public:
    // Constructor
    Smartpointer( T *ptr = NULL) {
        p = ptr;
    }
    // Destructors
    ~Smartpointer() {
        delete(p);
    }
    // Overloading dereferencing operator
    T & operator * () {
        return *p;
```

```
    }
};
int main() {
    Smartpointer<int> p(new int());
    *p = 26;
    cout<<"Value is: "<<*p;
    return 0;
}
```

Code - 6 Use abstract classes and polymorphism in C++ for calculating the areas of various shapes

```
#include <cmath> // for M_PI

// Abstract base class
class Shape {
public:
    // Pure virtual function providing interface framework.
    virtual double area() const = 0;

    // Virtual destructor to ensure proper cleanup of derived classes
    virtual ~Shape() {}
};

// Derived class for Circle
class Circle : public Shape {
private:
    double radius;
public:
```

```
    Circle(double r) : radius(r) {}

    // Override area() for Circle
    double area() const override {
        return M_PI * radius * radius;
    }
};

// Derived class for Rectangle
class Rectangle : public Shape {
private:
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}

    // Override area() for Rectangle
    double area() const override {
        return width * height;
    }
};

// Derived class for Triangle
class Triangle : public Shape {
private:
    double base, height;
public:
```

```
    Triangle(double b, double h) : base(b), height(h) {}

    // Override area() for Triangle
    double area() const override {
        return 0.5 * base * height;
    }
};

int main() {
    // Create objects of derived classes
    Shape* shapes[] = {
        new Circle(7.0),
        new Rectangle(4.0, 6.0),
        new Triangle(9.0, 10.0)
    };

    // Calculate and display the area of each shape
    for (Shape* shape : shapes) {
        std::cout << "Area: " << shape->area() << std::endl;
    }

    // Clean up
    for (Shape* shape : shapes) {
        delete shape;
    }
```

Code - 7 Inventory Management System:

Problem: Design a system to manage inventory for various products. Each product might have different attributes (name, price, quantity) and potentially unique functionalities (e.g., perishable items with an expiry date).

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

// Abstract base class for products
class Product {
protected:
    string name;
    double price;
    int quantity;

public:
    Product(const std::string& name, double price, int quantity)
        : name(name), price(price), quantity(quantity) {}

    virtual ~Product() {}

    virtual void display() const = 0;  // Pure virtual function to display product details

    // Common functions for all products
    virtual string getName() const {
        return name;
```

```
    }

    virtual double getPrice() const {
        return price;
    }
    virtual int getQuantity() const {
        return quantity;
    }
    virtual void setQuantity(int qty) {
        quantity = qty;
    }
};
// Derived class for perishable items
class PerishableItem : public Product {
private:
    string expiryDate;

public:
    PerishableItem(const std::string& name, double price, int quantity, const std::string& expiryDate)
        : Product(name, price, quantity), expiryDate(expiryDate) {}

    void display() const override {
        cout << "Perishable Item: " << name << "\nPrice: $" << price << "\nQuantity: " << quantity <<
"\nExpiry Date: " << expiryDate << std::endl;
    }
    string getExpiryDate() const {
        return expiryDate;
```

```
    }};
// Derived class for non-perishable items
class NonPerishableItem : public Product {
public:
    NonPerishableItem(const std::string& name, double price, int quantity)
        : Product(name, price, quantity) {}

    void display() const override {
  cout << "Non-Perishable Item: " << name << "\nPrice: $" << price << "\nQuantity: " << quantity << std::endl;
    }
};

// Inventory Management class
class Inventory {
private:
    vector<Product*> products;

public:
    ~Inventory() {
        for (Product* product : products) {
            delete product;
        }
    }
    void addProduct(Product* product) {
        products.push_back(product);
    }
```

```
    void displayInventory() const {
        for (const Product* product : products) {
            product->display();
            cout << std::endl;
        }
    }
    // Function to find a product by name
    Product* findProduct(const std::string& name) const {
        for (Product* product : products) {
            if (product->getName() == name) {
                return product;
            }
        }
        return nullptr;
    }
    // Function to update quantity of a product
    void updateQuantity(const std::string& name, int quantity) {
        Product* product = findProduct(name);
        if (product) {
            product->setQuantity(quantity);
        }
    }
};

int main() {
    Inventory inventory;
```

```
    // Add some products to inventory
    inventory.addProduct(new PerishableItem("Milk", 2.99, 30, "2024-07-15"));
    inventory.addProduct(new NonPerishableItem("Rice", 1.50, 100));

    // Display inventory
    inventory.displayInventory();

    // Update quantity of a product
    inventory.updateQuantity("Rice", 120);

    // Display inventory again to see the updated quantity
    cout << "Updated Inventory:" << std::endl;
    inventory.displayInventory();
    return 0;
}
```