/***

1. Access Control and Getters:

Create the User class with private members for username and profile picture (string).
Implement public member functions for the constructor and getters (accessor methods) for username and profile picture.

***/

```
#include<iostream>
#include<string>
using namespace std;
class User{
private:
   string username;
   string profilePicture;
public:
   User(string uname, string profilePic) {
      username=uname;
      profilePicture=profilePic;
   }
   string getUsername() {
      return username;
   }
   string getProfilePicture() {
      return profilePicture;
   }
};
int main() {
   User user1("sunny kumar" ,"image");
   cout<<"Username is: "<<user1.getUsername()<< endl;
   cout<<"Profile Picture is: "<< user1.getProfilePicture();
   return 0;
}
```

/***

2. Post Class and Display:

Create the derived class Post inheriting from User.
Add private members for post content (string) and timestamp (date/time format of your choice).
Implement a public member function getPostInfo that returns a formatted string containing username, profile picture, post content, and timestamp.

```
**********************************************************************************/

#include <iostream>
#include <string>
using namespace std;
class User {
private:
   string username;
   string profilePicture;
public:
   User(string uname, string profilePic) {
      username=uname;
      profilePicture=profilePic;
   }
   string getUsername() const {
      return username;
   }
   string getProfilePicture() const {
      return profilePicture;
   }
};
class Post:public User {
private:
   string postContent;
   string timestamp;
public:
   Post(string uname,string profilePic,string content,string time):User(uname, profilePic) {
      postContent = content;
      timestamp = time;                // Manual timestamp
   }
   string getPostInfo() const {
      return "Username is: " + getUsername() + "  "
            + "Profile Picture is: " + getProfilePicture() + "  "
            + "Post Content: " + postContent + "  "
            + "Timestamp: " + timestamp + "  ";
   }
};
int main() {
   Post post1("sunny kumar sharma","image","india won the T-20 world cup","2024-07-02 ");
   cout << post1.getPostInfo();
   return 0;
}

/**********************************************************************************
```

Define a friend function basicInteract that takes two User objects (or derived class objects) as arguments.
Inside the function, simply print a generic message like "User1 interacts with User2."

**********/

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

class User {
private:
    string username;
    string profilePicture;

public:
    User(string uname,string profilePic) {
        username=uname;
        profilePicture=profilePic;
    }

    string getUsername() const {
        return username;
    }

    string getProfilePicture() const {
        return profilePicture;
    }

    friend void basicInteract(const User& user1,const User& user2);
};

void basicInteract(const User& user1,const User& user2) {      // Friend function definition

    cout<<user1.getUsername()<<"interacts with "<<user2.getUsername()<<endl;
}

int main() {
    User user1("sunny kumar","image1");
    User user2("raju", "image2");

    basicInteract(user1, user2);

    return 0;
```

```
}

/*******************************************************************************
Create overloaded versions of the interact function:
likePost(User& user, Post& post): This function should print a message indicating the user liked
the post.
followUser(User& follower, User& followed): This function should print a message indicating the
user started
following another user.

*******************************************************************************/
#include <iostream>
#include <string>
using namespace std;
class User;
class Post;
void interact(User& user, Post& post);
void interact(User& follower, User& followed);
class User {
private:
    string username;
    string profilePicture;
public:
    User(string uname, string profilePic) {
        username = uname;
        profilePicture = profilePic;
    }
    string getUsername() const {
        return username;
    }
    string getProfilePicture() const {
        return profilePicture;
    }
    friend void interact(User& user, Post& post);
    friend void interact(User& follower, User& followed);
};
class Post {
private:
    string content;
public:
    Post(string text) {
        content = text;
    }
```

```
    string getContent() const {
        return content;
    }
    friend void interact(User& user, Post& post);
};
void interact(User& user, Post& post) {
    cout << user.getUsername() << " liked the post: \"" << post.getContent() << "\"" << endl;
}

void interact(User& follower, User& followed) {
    cout << follower.getUsername() << " started following " << followed.getUsername() << endl;
}

int main() {
    User user1("rohit sharma","profile1");
    User user2("virat kohli", "profile2");
    Post post("Check out my new post");
    interact(user1, post);
    interact(user2, user1);

    return 0;
}

/*****************************************************************************

static keyword uses

*****************************************************************************/
#include <iostream>
using namespace std;
class myclass{
    private:
    static int counter;
    public:
    myclass()
    {
        counter++;
    }
    static int getcount(){
        return counter;
    }
};
int myclass::counter=0;
int main()
```

```
{
   myclass obj1;
   myclass obj2;
   myclass obj3;
   cout<<"no. of object created is: "<<myclass::getcount<<endl;
   return 0;
}
```

***

```
#include <iostream>
using namespace std;
class myclass {
private:
   int counter;
public:
   myclass() {
      counter++;
   }
   int getcount(){
      return counter;
   }
};
int main() {
   myclass obj1;
   myclass obj2;
   myclass obj3;
   cout <<"objects created for obj1 is: "<<obj1.getcount() << endl;
   cout <<"objects created for obj2is: "<<obj2.getcount() << endl;
   cout <<"objects created for obj3 is: "<<obj3.getcount();
   return 0;
}
```

*****

```
/*****************************************************************************

static keyword uses

*****************************************************************************/
#include <iostream>
using namespace std;
class myclass{
```

```
    private:
    static int counter;
    int count;
    public:
    myclass()
    {
        counter++;
        count++;
    }
    static int getcounter(){
        return counter;
    }
    int getcount()
    {
        return count;
    }
};
int myclass::counter=0;
int main()
{
    myclass obj1;
    myclass obj2;
    myclass obj3;
    cout<<"no. of object created is: "<<myclass::getcounter<<endl;
    cout<<"no. of object created is: "<<obj1.getcount()<<endl;
    cout<<"no. of object created is: "<<obj2.getcount()<<endl;
    cout<<"no. of object created is: "<<obj3.getcount()<<endl;
    return 0;
}
```

/*
Distance Converter:
Create a class named DistanceConverter. Include the following static methods:

convertMilesToKm(double miles): Converts miles to kilometers (1 mile = 1.60934 kilometers).
convertKmToMiles(double kilometers): Converts kilometers to miles. In your main function, prompt the user for a distance and a unit (miles or kilometers). Use the appropriate static method from the DistanceConverter class to perform the conversion and display the result to the user.
Math Utility Class:
Design a class named MathUtil. Include static methods for basic mathematical operations:

add(int a, int b): Adds two integers.
subtract(int a, int b): Subtracts two integers.
multiply(int a, int b): Multiplies two integers.
divide(int a, int b) (optional): Divides two integers with error handling for division by zero. In your main function, prompt the user for two numbers and an operation (+, -, *, or /). Use the corresponding static method from the MathUtil class to perform the calculation and display the result.
*/

```
#include <iostream>
using namespace std;
class DistanceConverter {
public:
    static double convertMilesToKm(double miles) {
        return miles * 1.60934;
    }
    static double convertKmToMiles(double kilometers) {
        return kilometers / 1.60934;
    }
};
class MathUtil {
public:
    static int add(int a,int b) {
        return a+b;
    }

    static int subtract(int a,int b) {
        return a-b;
    }

    static int multiply(int a,int b) {
        return a*b;
    }

    static double divide(int a,int b) {
        if (b==0) {
            cout<<"not divisible by zero"<<endl;
            return 0;
        }
        return static_cast<double>a/b;
    }
};

int main() {
    double distance;
```

```
    char unit;

    cout<<"Enter a distance: ";
    cin>>distance;

    cout<<"Enter unit (miles ke liye m dbaye,kilometers ke liye k dbaye): ";
    cin>>unit;

    switch (unit) {
        case 'm':
            cout<<distance<<"miles is equal to
"<<DistanceConverter::convertMilesToKm(distance)<<"kilometers"<<endl;
            break;
        case 'k':
            cout<<distance<<"kilometers is equal to
"<<DistanceConverter::convertKmToMiles(distance)<<"iles"<<endl;
            break;
        default:
            cout<<"Invalid unit enter"<<endl;
            break;
    }
    int num1,num2;
    char operation;

    cout<<"Enter two numbers: ";
    cin>>num1>>num2;

    cout<<"Enter operation(+, -, *, /): ";
    cin>>operation;

    switch (operation) {
        case '+':
            cout<<"Result of "<<num1<<" + "<<num2<<" = "<<MathUtil::add(num1,num2)<<endl;
            break;
        case '-':
            cout<<"Result of "<<num1<<" - "<<num2<<" =
"<<MathUtil::subtract(num1,num2)<<endl;
            break;
        case '*':
            cout<<"Result of "<<num1<<" * "<<num2<<" = "<<MathUtil::multiply(num1,num2)<<endl;
            break;
        case '/':
            cout<<"Result of "<<num1<<" / "<<num2<<" = "<<MathUtil::divide(num1, num2)<<endl;
            break;
```

```
        default:
            cout<<"Invalid operation"<<endl;
            break;
    }

    return 0;
}
```

/*Simple Currency Converter:
Create a class named CurrencyConverter. Define a static variable named exchangeRate (e.g., USD to EUR exchange rate). Implement static methods:

convertToEur(double amount): Converts an amount from the base currency (USD) to EUR based on the exchange rate.
convertFromEur(double amount): Converts an amount from EUR to the base currency (USD). In your main function, prompt the user for an amount and a conversion operation (USD to EUR or EUR to USD). Use the appropriate static method from the CurrencyConverter class to perform the conversion and display the result.
*/

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

class CurrencyConverter {
private:
    static double exchangeRate;
public:
    static double convertToEur(double amount) {
        return amount*exchangeRate;
    }

    static double convertFromEur(double amount) {
        return amount/exchangeRate;
    }
};
double CurrencyConverter::exchangeRate = 0.85; //  1 Usd = 0.85 Eur

int main() {
    double amount;
    char operation;
    cout<<"Enter an amount: ";
    cin>>amount;
```

```
    cout<<"Enter conversion (U for USD to EUR, E for EUR to USD): ";
    cin>>operation;
    switch (operation) {
        case 'U':
            cout<<amount<<"USD is equal to
"<<CurrencyConverter::convertToEur(amount)<<"EUR"<<endl;
            break;
        case 'E':
            cout<<amount<<"EUR is equal to
"<<CurrencyConverter::convertFromEur(amount)<<"USD"<<endl;
            break;
        default:
            cout<<"Invalid operation"<<endl;
            break;
    }
    return 0;
}
```

template in C++

[1]

```
#include <iostream>
using namespace std;
template<class T>
T add(T &a, T &b) {
    T result = a + b;
    return result;
}
int main() {
    int i = 2;
    int j = 3;
    float m = 2.3;
    float n = 1.2;

    cout << "Addition of i and j is: " << add(i, j);
    cout << '\n';
    cout << "Addition of m and n is: " << add(m, n);

    return 0;
}
```

[2]

```
#include <iostream>

using namespace std;

template<class X, class Y>
void fun(X a, Y b) {
   std::cout << "Value of a is : " << a << std::endl;
   std::cout << "Value of b is : " << b << std::endl;
}

int main() {
   fun(15, 12.3);

   return 0;
}
```

[3]

```
#include <iostream>

using namespace std;

template<class X>
void fun(X a) {
   std::cout << "Value of a is : " << a << std::endl;
}

template<class X, class Y>
void fun(X b, Y c) {
   std::cout << "Value of b is : " << b << std::endl;
   std::cout << "Value of c is : " << c << std::endl;
}

int main() {
   fun(10);
   fun(20, 30.5);

   return 0;
}
```

Design a function template named compare that takes two arguments of the same type and returns a boolean value indicating whether the first argument is greater than, less than, or equal to the second argument. How would you adapt this template to work with custom data types?

```
#include <iostream>
#include <string>

using namespace std;

class Person {
public:
    string name;
    int age;

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

    bool operator>(const Person& other) const {
        return age > other.age;
    }

    bool operator<(const Person& other) const {
        return age < other.age;
    }

    bool operator==(const Person& other) const {
        return age == other.age;
    }
};

template <typename T>
int compare(const T& a, const T& b) {
    if (a > b) {
        return 1;
    } else if (a < b) {
        return -1;
    } else {
        return 0;
    }
}

int main() {
    Person p1("Alice", 25);
    Person p2("Bob", 30);
```

```
    cout << compare(p1, p2) << endl;  // Output: -1

    Person p3("Charlie", 25);
    cout << compare(p1, p3) << endl;  // Output: 0

    return 0;
}
```

Implement a function template named swap that exchanges the values of two variables of the same type. Discuss the potential limitations of this approach when dealing with complex data structures.

```
#include <iostream>

using namespace std;

template <typename T>
void swap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

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

    string str1 = "sunny", str2 = "sharma";
    cout << "Before swap: str1 = " << str1 << ", str2 = " << str2 << endl;
    swap(str1, str2);
    cout << "After swap: str1 = " << str1 << ", str2 = " << str2 << endl;

    return 0;
}
```

Consider a scenario where you need to find the minimum value in an array. Create a function template named findMin that works with any data type for which the comparison operator (<) is defined. Explain how function templates promote code reusability in this case.

```
#include <iostream>

using namespace std;

template <typename T>
T findMin(const T arr[], int size) {
    T minValue = arr[0];
    for (int i = 1; i < size; ++i) {
        if (arr[i] < minValue) {
            minValue = arr[i];
        }
    }
    return minValue;
}

int main() {
    int intArr[] = {5, 2, 9, 1, 5, 6};
    int intSize = sizeof(intArr) / sizeof(intArr[0]);
    cout << "Minimum int value: " << findMin(intArr, intSize) << endl;

    double doubleArr[] = {3.5, 2.1, 9.8, 1.3, 5.4, 6.7};
    int doubleSize = sizeof(doubleArr) / sizeof(doubleArr[0]);
    cout << "Minimum double value: " << findMin(doubleArr, doubleSize) << endl;

    string strArr[] = {"apple", "orange", "banana", "pear"};
    int strSize = sizeof(strArr) / sizeof(strArr[0]);
    cout << "Minimum string value: " << findMin(strArr, strSize) << endl;

    return 0;
}
```