Question 1
What is shared memory and when will you use it?

Answer to the question 1:

Shared memory is a highly efficient way of sharing data between running processes or programs. It allows two or more unrelated processes to access the logical memory segment. Sharing a common piece of the memory segment is the fastest way that IPC can be achieved

For example, Program A provides a list to Program B, it saves the data in shared memory and marks it with a semaphore or other flagging system to signal that it is ready to be read by Program B. When Program B finds the file, it checks the semaphore to see if it is allowed to touch that file. If allowed, then it does what it needs to do to the file, puts it in shared memory or updates it.

Question 2:
What are the restrictions, if any, in C# OOP?

Answer to the question 2:
C# doesn't support multiple-inheritance, however we can use interfaces to achieve this feature.

Question 3:
Give an example, in your own words, of polymorphism and when it should be used

Answer to the question 3:
The word polymorphism is derived from the Greek word, where Poly means many and morph means faces/ behaviors. So, the word polymorphism means the ability to take more than one form. It means single name & multiple meaning & whenever we try to access that particular name there must be a binding between that call & between that function. So there are two types of binding to call that function. Those are compile time binding and run time binding. Technically we can say that when a function shows different behaviors when we passed different types and numbers of input values, then it is called Polymorphism in C#. So behaving in different ways depending on the input received is known as polymorphism i.e. whenever the input changes, automatically the output or the behavior also changes. It gets useful when you have different types of objects and can write classes that can work with all those different types because they all adhere to the same API.

C# support 2 types of polymorphism. Those are Run time polymorphism and compile time polymorphism.

For example of run time polymorphism, think of a base class called Animal that has a method called animalSound(). Derived classes of Animals could be Cow, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the cow hamba, and the cat meows, etc.)

```
Public class Animal  // Base class (parent)
{
   public virtual void animalSound()
   {
      Console.WriteLine("The animal makes a sound");
   }
}

class Cow: Animal  // Derived class (child)
{
   public override void animalSound()
   {
      Console.WriteLine("The Cow says: Hamba Hamba");
   }
}

class Dog : Animal  // Derived class (child)
{
   public override void animalSound()
   {
      Console.WriteLine("The dog says: bow wow");
   }
}

class Program
{
   static void Main(string[] args)
   {
      Animal myAnimal = new Animal();  // Create a Animal object
      Animal myCow = new Cow();  // Create a Cow object
      Animal myDog = new Dog();  // Create a Dog object

      myAnimal.animalSound();
      myCow.animalSound();
      myDog.animalSound();
   }
}
```

In the above example, we can see the virtual function animalSound bound at runtime.

For example of compile time polymorphism, It is also known as Early Binding. Method overloading is an example of Static Polymorphism. In overloading, the method / function has the same name but different signatures. It is also known as Compile Time Polymorphism because the decision of which method is to be called is made at compile time. Overloading is the concept in which method names are the same with a different set of parameters.

In the following example, a class has two methods with the same name "Add" but with different input parameters (the first method has three parameters and the second method has two parameters).

```
public class TestData
{
    public int Add(int a, int b, int c)
    {
        return a + b + c;
    }
    public int Add(int a, int b)
    {
        return a + b;
    }
}
class Program
{
    static void Main(string[] args)
    {
        TestData dataClass = new TestData();
        int add2 = dataClass.Add(45, 34, 67);
        int add1 = dataClass.Add(23, 34);
    }
}
```

Question 4:
Please provide a practical example of interface implementation in C#, describing exactly what benefits you would achieve through your design.

Answer to the question 4:

This example strictly follows the Interface segregation principle rules of SOLID principles.

```
public interface IBaseWorker
{
    string ID { get; set; }
    string Name { get; set; }
    string Email { get; set; }
}

public interface IFullTimeWorkerSalary : IBaseWorker
{
    float MonthlySalary { get; set; }
    float OtherBenefits { get; set; }
    float CalculateNetSalary();
}

public interface IContractWorkerSalary : IBaseWorker
{
    float HourlyRate { get; set; }
    float HoursInMonth { get; set; }
    float CalaculateWorkedSalary();
}

public class FullTimeEmployeeFixed : IFullTimeWorkerSalary
{
    public string ID { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public float MonthlySalary { get; set; }
    public float OtherBenefits { get; set; }
    public float CalculateNetSalary() => MonthlySalary + OtherBenefits;
}
```

```
public class ContractEmployeeFixed : IContractWorkerSalary
{
    public string ID { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public float HourlyRate { get; set; }
    public float HoursInMonth { get; set; }
    public float CalaculateWorkedSalary() => HourlyRate * HoursInMonth;
}
```

In the above code, we see that we have created a base interface which has the common properties of all employees. Then, we created two more interfaces which implement the base interface and these interfaces are for the full time and contract employee, respectively. They only contain properties and methods for the particular type of employee. Finally, in our classes for the full time and contract employee, we only implement the required interface (full time or contract). No additional method or property is required.

Benefits of Interface

- Achieve the multiple inheritance
- Achieve loose coupling code
- Provide abstraction behaviors
- Implement Dependency injection
- Higher level components don't dependent on lower level component and vice versa

Question 5:
In your experience what aspects of object oriented development has yielded the most maintainable, versatile, flexible solution. Provide examples where possible.

Answer to the question 5:

- Modularity for easier troubleshooting and development
- We can build the programs from standard working modules that communicate with one another, rather than having to start writing the code from scratch which leads to saving of development time and higher productivity,
- OOP language allows to break the program into bit-sized problems that can be solved easily (one object at a time).
- The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost.

- OOP systems can be easily upgraded from small to large systems.
- It is possible that multiple instances of objects co-exist without any interference,
- It is very easy to partition the work in a project based on objects.
- It is possible to map the objects in the problem domain to those in the program.
- The principle of data hiding helps the programmer to build secure programs which cannot be invaded by the code in other parts of the program.
- By using inheritance, we can eliminate redundant code and extend the use of existing classes.