Question 1 PART A

Pure Virtual Function:

A pure virtual Function is a Function which can be overridden in the derived class but cannot be defined. A virtual Function can be declared as pure by using the operator =0

example:

```
Virtual void Function1() // virtual, Not pure
Virtual void Function2() =0 // pure virtual.
```

Abstract Base Classes:

An abstract baseclass (ABC) is a class that cannot be instantiated on its own. It's designed to serve as a blueprint for derived classes, ensuring they implement certain common functionalities.
The key characteristic of an ABC is that it contain at least one pure virtual Function

Interfaces:

A special kind of Abstract class where all Functions are Pure virtual.

Key roles in class Hierachies:

1) Defining a core Foundation:

Establish a shared foundation for related classes, defining common properties and

method.
Promote code reusability by consolidating shared functionality within the abstract class.

2) Enforcing Consistency:
Mandate the implementation of specific methods in derived classes, ensuring adherence to a defined interface.
Facilitate polymorphism, enabling code to operate on objects of different derived classes through a common interface.

3) Structuring Complex Hierarchies:
Organize complex class relationships effectively, promoting a clear and logical structure.
Define intermediate levels of abstraction, breaking down large hierarchies into manageable components.

4) Facilitating Design Patterns:
Serve as cornerstone for various design patterns, such as the template Method pattern &A
Enable the creation of flexible and reusable design solution.

Question 1 PART C

Association:

Association represents a relationship between two or more classes where objects of one class are connected or interact with objects of another class.

Real world example:

Classes: Person and Address

A person has an address. The association here reflects the fact that a person is connected to an address

```
Class Address {
 Public :
     std :: string city;
     std :: string zipcode;
Address(std :: string city , std :: string zipcode) :
  City(city); zipcode(zipcode) {}
};
Class Person {
 public :
       std :: string name;
       Address * address;  // Association with address

 Person(std :: string name) : name(name); address
  (nullptr) {}
};
```

```
int main () {
Address * johnAddress = new Address ("cityville", "12345");
person sharan("sharan");
  // Associating the person with an address
  sharan.address = ~~john~~ sharanAddress;
                    sharan // Accessing Associated Address
std :: cout << ~~john~~.name << "s address: <<
  sharan.address -> city << ", " << sharan.address
  -> zipcode << std :: endl;
  // clean up memory
  delete sharanaddress;

  return 0;
}
```

**Aggregation:**

Aggregation is a type of association where one class represents a whole and another class represents a part but the part can exist independely of the whole.

**Real world example:**

Classes University and Department.

A university is composed of multiple department if the university is closed the department can still exist independly.

```
Class Department {
Public:
 Std::string name;
Department(std::string name) : name(name){}
 };
Class University {
public:
 Std:: String name;
 Std:: vector<Department*> departments;
    // Aggregation with Department
 University (Std:: string name) : name(name){}
   // Method to add a department to the University
  void adddepartment* department) {
     department.push_back(department);
   }
};
 int main(){
 University (custUniversity) ("CustUniversity");
 Department computerscience("Computer Science");
 Department Physics ("Physics");
   // Aggregation Department into the University.
 CustUniversity.addDepartment(&computerScience);
 CustUniversity.adddepartment(&physics);

for(const auto &department: custUniversity.departments)
 {
 Std::cout<<custUniversity.name <<" has a department:"
  << department -> name<< Std::endl;
 }
  return 0;
}
```

## Composition:

Composition is a stronger form of aggregation where the part class has a strong lifecycle dependency on the whole class. If the whole is destroyed the parts also destroyed.

## Real world example:

Classes car and engine
A car is composed of an engine. If the car is destroyed, the engine is also no longer usable.

```
class Engine{
Public:
    int horsepower;
    Engine(int horsepower):horsepower(horsepower){}
};
class Car{
public:
        std:: string make;
        std:: string model;
        Engine* engine; // Composition with engine

Car(std::string make, std::string model, int horsepower)
 : make(make), model(model), engine(new Engine
   (horsepower)){}

  // Destruct to clean memory
  ~Car(){
   delete engine;
```

```
};

int main() {
Car myCar("Toyota", "Alto", 1000);

// Accessing the composition
std::cout << "My car has an engine with " <<
myCar.engine->horsePower << " horsepower."
<< std::endl;

return 0;
}
```

**Question 2**

UML Class Diagram