Date:.....

Item 22 : Declare Data members private.

Why not public data members?

⇒ (1) If data members are public, everybody has read write access to it.

We can implement no access, read only access, write read access using get & set functions.

```
Class AccessLevels {
    public:
        int getReadOnly() const { return readOnly; }
int setReadWrite (int value) { readWrite = value; }
int getReadWrite() const { return readWrite; }
    private:
        int noAccess;
        int readOnly;
        int readWrite;
};
```

Date:

② Encapsulation.

Suppose we have a class SpeedDataCollection where we have a funct averageSoFar.

averageSoFar can be implemented in 2 ways.

① keep a running average → but this makes SpeedDataCollection object bigger

② averageSoFar can be implemented such that it returns / recalculates the value of running average each time → this makes it slower.

⇒ On m/c where memory is tight (roadside device) & where averages are needed infrequently computing average each time is better.

⇒ In application where average are needed frequently & memory is not an issue keeping running average is preferable.

Now the point is ⇒ by accessing the average thru a member funct (encapsulating) we can interchange these implementation, client will only need to recompile.

Date:.....

we have the right to change implementation but if done so, how much code client has to change?

In case of public data members, too much client code will be broken also in protected, all the derived classes that uses it which is unknowably large amt of code.

Hence protected are as unencapsulated as public data members.

Item 23 explains → something is encapsulated is inversely proportional to amt of code that might be broken

⇒ Hence declare data members private