C++ Assignment -1

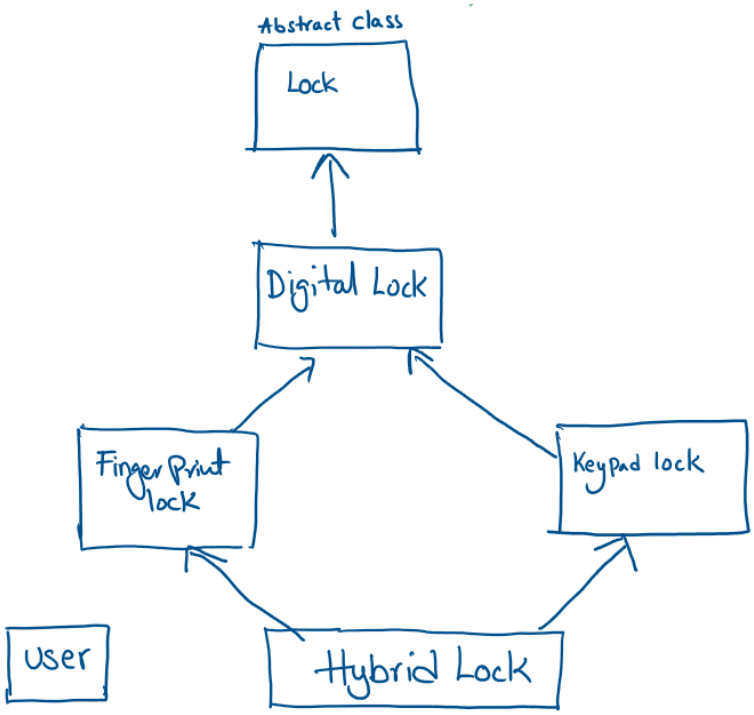
The Task:

Your first assignment is to write a single body of code example that integrates and demonstrates the following aspects of OOP in C++:

1. **Inheritance** through the use of multiple classes. This should include multiple inheritance as one example, which does not have to make perfect sense in the real-world. (Please note Point 2.)
2. **Separate compilation** with all classes in your assignment (i.e., all classes should have .cpp and .h files).
3. **One abstract class** with an abstract method that is implemented in a child class. This should be integrated into your code in Point 1.
4. Correct and appropriate use of the **access specifiers** public, private, and protected with your classes.
5. **Passing an object** to a function by value and by reference. Demonstrate the impact of the change on the states of your object.
6. Calling a method on an object that is **const qualified** and passing the same object to a function by **constant reference**.
7. Correct use of **new and delete** for the allocation of an object/objects, with operations on the object using pointers.
8. Create an **array of pointers to objects** (minimum 3 elements) of one of four classes and pass this array to a function that displays all elements.
9. A simple **destructor** with some basic functionality in one of your classes. Demonstrate two examples of how it can be called.
10. Use of **dynamic binding** with virtual & non-virtual methods. Demonstrate the impact.
11. Correct use of the + and == **overloaded operators** for one of your classes.
12. **Overload the assignment operator** (=) for your class and demonstrate that it works correctly for chain operations.
13. Write code to demonstrate an **object passing itself** to a function (that is not part of a class), which modifies the state of that object.
14. A class with a **modified copy constructor** and demonstrate the effect of this on pass-by-value and pass-by-reference calls.
15. Use of two different C++ **explicit style casts.**
16. Implement **static states** and **methods** in one of your classes and show example usage and impact.
17. Implement a **non-member operator** for one of your classes.
18. Create a **vector** container to contain a number of objects of one of your classes.
19. Use an **algorithm** on your container using the examples in the notes.
20. Create a simple **Smart Pointer** to an object of one of your classes and demonstrate it in action.

This assignment is based on security System concept, it contains a variety of Digital Locks. They are Key pad Lock, Finger print lock and a Hybrid lock which takes both pin and fingerprint as input to authenticate the any User. This Application contains classes for each Lock and a Main class with a main function.

Structure of



the classes

Please find the implementations directly by searching with keyword Point __ **ex: Point 11**

1. **Inheritance** is satisfied by inheriting **DigitalLock** in **FingerPrintLock** and **keyPadLock** classes and Inheritance through the use of **multiple classes** is done by inheriting **FingerPrintLock** and **KeyPadLock** in **HybridLock**.

2. All the classes are created with separate **.cpp** and **.h** files are compiled separately .I have added **declarations** in **.h** files and **definitions** in **.cpp** files.
3. Created an abstract class **Lock** with 2 pure virtual functions and implemented those in inherited **DigitalLock** class
4. Used **access specifiers** properly throughout the application. Can be verified in all **Class.h** files
5. In the **MainClass.cpp** file Under **Point 5**, I have implemented **call by value** and **call by reference.** There a user Lucas is trying to access the hybridlock, when he doesn't have access, so he tried to open by masking himself as owner(vishnu). Firstly Lucas enters a method **userMaskPlanA()** by value, and that doesn't work and later he passes to **userMaskPlanB()** then he can successfully able to mask himself as owner
6. Not Implemented
7. Correct use of **new** and **delete**. In the **MainClass.cpp** under Point 7, created a pointer object and accessed a lock and called **delete** object; to de allocate the memory used by the pointer object.
8. In the **MainClass.cpp** under Point 8, I have created an array of user objects with a fixed size 3 and called a **displayUserDetails()** function to display all it elements.
9. Destructor with basic functionality is satisfied by adding some functionality in **User class** destructor under **Point 9**. In general a Destructor is called automatically after our object goes out of scope. However we can also call it from the object.
10. In the **MainClass.cpp** under **Point 10**, I have called **printLockType()** which calls another dynamically overridden method **getLockType()** which prints type of lock. Here since I have declared **getLockType()** function as **virtual** it calls the **getLockType()** method based on the type of object otherwise it will call only the parent class method(DigtalLock) and not child class method (KeypadLock).
11. **+ and == overloaded operators** is implemented in **HybridLock.** Please find the implementations with word **Point 11** in the assignment. Under point 11 there are different hybrid locks created with different security levels. The function **checkUserAndDirect()** checks the **user** and **security level** of the locks and direct user to the lock which has high security Level. And for owner(vishnu) it checks whether the chosen lock has higher security level than appleCatalina HybridLock.
12. In the **MainClass.cpp** under **Point 12,** in the same **checkUserAndDirect() function object windows** is assigned to **chosenLock and androidPie** as a chain operation there the assignment operator of HybridLock gets called and works properly.
13. It is implemented in the **User.cpp** class. Whenever any user object tries to **printUserDetails(),** the object ref with keyword **this** will be sent to an external method and there the name gets changed. Thus passing object itself to a function which modifies the state of object satisfies.
14. **Modified copy constructor** is implemented in the **HybridLock class**. Here when we copy an another hybridLock it copies the **HybridLock** class object but it decreases the security level by one. Please check the copying implementation in main class under **Point 14**.
15. One C++ explicit type cast is implemented in **MainClass.cpp** file.
16. **Static variables** and a **method** is implemented in **DigitalLock** class. The **informPolice() static method** gets called when the lock fails to open and other **static states positiveLight** and **negativeLight** gets called in **FingerPrintLock.cpp** and **DigitalLock.cpp openLock()** method.
17. Not Implemented
18. In the **MainClass.cpp** under **Point 18,** I have created a **vector container** of 3 Hybrid Lock's and used an algorithm to display the elements of them.
19. In the **MainClass.cpp** under **Point 19,** I have used the algorithm **for_each** and **count** for a vector of Hybridlock Objects.

20. In the **MainClass.cpp** under **Point 20,** I have implemented a **Smart Pointer** of type **unique_ptr.** Here I **cannot copy** the object of lock created by the **unique_ptr**. But I **can only move** the pointer to another **unique pointer** object.

Classes are listed in the order below

1. MainClass
2. Lock
3. DigitalLock
4. KeyPadLock
5. FingerPrintLock
6. Hybridlock
7. User

MainClass.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <memory>
#include "Lock.h"
#include "User.h"
#include "DigitalLock.h"
#include "KeyPadLock.h"
#include "FingerPrintLock.h"
#include "HybridLock.h"
using std::cout;
using std::endl;
using std::vector;
using std::unique_ptr;
using std::count;



void userMaskPlanA(User x) {
//Point 5:
//this doesn't work because any changes happen to the x will be here in
scope
//and cannot reflect in other place. Only in case of returning the object
we can make use of this method
	if (x.name != "vishnu") {
		x.name = "vishnu";
	}
}




void userMaskPlanB(User &x) {
```

```
//Point 5:
//since the user enters by reference, whatever changes we make to the
object will apply.
	if (x.name != "vishnu") {
		x.name = "vishnu";
	}
}

bool checkPower(HybridLock a, HybridLock b) {
//Point 11
//	overloaded > and == operators
	if (a > b || a == b) {
		cout << "true" << endl;
		return true;
	} else {
		cout << "false " << endl;
		return false;
	}
}

void checkUserAndDirect(User bankUser, HybridLock choosenLock,
		HybridLock androidKitkat, HybridLock appleCatalina,
		HybridLock androidPie, HybridLock windows) {
	if (bankUser.name == "vishnu" && (choosenLock > appleCatalina)) {
		cout << "User is vishnu" << endl;
		cout << "choosenLock is greater than appleCatalina" << endl;
		choosenLock.openLock(bankUser);
	} else {
		cout << "choosenLock is NOT greater than appleCatalina" <<
endl;

//Point 12 overloaded assignment(=) operator for chain operations
		choosenLock = androidPie = windows;
//The overloaded operator chain operations properly works so the
securitylevel value here fall from 11 to 9
cout << "securityLevel: of " << choosenLock.lockOS << " is
"<<choosenLock.securityLevel << endl;
		HybridLock hyWindowsEleven = choosenLock + windows;
		checkPower(hyWindowsEleven, appleCatalina); //true
		hyWindowsEleven.openLock(bankUser);
	}
}

void display(HybridLock l) {
	cout << "l.lockOS: " << l.lockOS << endl;
	cout << "l.securityLevel: " << l.securityLevel << endl;
}

void displayUserDetails(User x) {
	cout << "User name: " << x.name << endl;
	cout << "User pin : " << x.pin << endl;
	cout << "User fingerPrint : " << x.fingerPrint << endl;
}




int main() {
```

```
//    Point 13 passing object itself to a fun implemented in User Class.
      User robot = User("robot");
      robot.printUserDetails();

      //Point 5: call by value and call by reference
      HybridLock hybridLock = HybridLock("hylock", 10);
      User lucas = User("lucas", 1234, "fingerprint");
      userMaskPlanA(lucas); //access fails because lucas cannot mask as
vishnu
      hybridLock.openLock(lucas);
      userMaskPlanB(lucas); //lucas got access because he changed his
name to vishnu by reference
      hybridLock.openLock(lucas);

      //Point 10
      //dynamic binding virtual methods
      DigitalLock dlock;
      dlock.printLockType();
      KeyPadLock klock;
      klock.printLockType();

      //    Point 11 & Point 12
      HybridLock appleCatalina = HybridLock("catalina", 11);
      HybridLock appleMontery = HybridLock("Montery", 12);
      HybridLock androidKitkat = HybridLock("kitkat", 5);
      HybridLock androidPie = HybridLock("pie", 7);
      HybridLock windows = HybridLock("windows10", 10);
      User bankUser = User("vishnu", 1234, "fingerprint");

      checkUserAndDirect(bankUser, androidKitkat, androidKitkat,
appleCatalina,
                  androidPie, windows); // androidKitkat is not greater
than appleCatalina

      checkUserAndDirect(bankUser, appleMontery, androidKitkat,
appleCatalina,
                  androidPie, windows); // appleMontery is greater than
appleCatalina

      //Point 14 Modified copy constructor
      HybridLock aa = HybridLock("catalina", 11);
      HybridLock bb(aa);
      cout << "aa.securityLevel = " << aa.securityLevel << endl;
      cout << " bb.securityLevel = " << bb.securityLevel << endl;

      HybridLock *cc = new HybridLock("testOS", 5);
//    security level wont fall by one for dd because we are passing by
reference
      HybridLock *dd(cc);
      cout << "cc.securityLevel = " << cc->securityLevel << endl;
      cout << " dd.securityLevel = " << dd->securityLevel << endl;

      //Point 18
      //created a vector for 3 Hybrid Locks and used a vector function
for_each to display its elements.
      vector<HybridLock> vectorHy { HybridLock("a", 2), HybridLock("a",
2),
                  HybridLock("b", 3), HybridLock("c", 4) };

      //    Point 19 used count and for_each algorithms
      int count = std::count(vectorHy.begin(), vectorHy.end(),
```

```
                  HybridLock("a", 2));
      cout << "COUNT=====" << count << endl;
      for_each(vectorHy.begin(), vectorHy.end(), display);

//Point 20 Smart pointer Unique
      unique_ptr<HybridLock> uniqueHyLock(
                  new HybridLock("Rare Custom lock OS", 100));

      cout << "unique ptr : " << uniqueHyLock->lockOS << endl;
//here i am not able to copy it as a normal HybridLock and i can only
copy it as a unique pointer
//HybridLock copiedUniquelock = move(uniqueHyLock); // Not possible
      unique_ptr<HybridLock> copiedUniquelock = move(uniqueHyLock);
      cout << "unique ptr Moved : " << copiedUniquelock->lockOS << endl;

// Point 15 Explicit casing. Digital lock is dynamically casted to
KeyPadlock.
      DigitalLock b = DigitalLock();
      b.printLockType();
      DigitalLock *a = new DigitalLock();
      a->printLockType();
      KeyPadLock *k = dynamic_cast<KeyPadLock*>(a);
      a->printLockType();
      k->printLockType();

//Point 8
//created array of pointers for a class User and displaying it elements
by passing to a function
      int i;
      User *testUsers[3];
      testUsers[0] = new User("name1", 12, "fp1");
      testUsers[1] = new User("name2", 13, "fp2");
      testUsers[2] = new User("name3", 14, "fp3");
      for (i = 0; i <= 3; i++) {
            displayUserDetails(*testUsers[i]);
      }

//    delete testUsers[3];
      //Point 7
      //accessing HybridLock &
      //Demonstrated 2 ways of calling a destructor,
      //one by calling destructor and another by using delete keyword.
      HybridLock dynamicLock = HybridLock("dynamicLock", 11);
      User *sunith = new User("sunith", 1234, "fingerprint");
      dynamicLock.openLock(*sunith);
      //cleaning dynamically allocated memory with delete,
      //so we cannot know what output it gives when we use the object
again after deallocating memory.
      delete sunith;
      dynamicLock.openLock(*sunith);
      //Point 9: calls the destructor of User which in reality doesn't
clear/ deallocate the object memory
      sunith->~User();

//.............................................................
// Below are the general lock accessing implementations
//    //accessing digital lock with owner
      DigitalLock digitalLock = DigitalLock();
      User *owner = new User("vishnu", 1234, "myfpdata");
      digitalLock.openLock(*owner);
```

```
        //accessing digital lock with random person/wrong password
        User *random = new User("simha", 12, "myfpdata");
        digitalLock.openLock(*random);

        // accessing fingerPrintLock
        FingerPrintLock fpLock;
        User *ajay = new User("Ajay", 1234, "fingerprint");
        fpLock.openLock(*ajay);

        // accessing KeyPadLock
        KeyPadLock *keypadLock = new KeyPadLock();
        User *john = new User("John", 1234);
        keypadLock->openLock(*john);

        //accessing HybridLock
        HybridLock lk = HybridLock("sas", 11);
        User one = User("one", 1234, "fingerprint");
        User two = User("two", 1234, "fingerprint");
        two = one;
        two.printUserDetails();
        one.printUserDetails();
        lk.openLock(one);

        return 0;
}
```

Lock.h

```
#ifndef LOCK_H_
#define LOCK_H_
#include <iostream>
#include "User.h"
using std::cout;

//Point 3: Abstract Class with pure virtual functions
class Lock {
public:
        virtual void openLock(User user)= 0;
        virtual bool getLockState() =0;
        Lock();
        virtual ~Lock();
};

#endif /* LOCK_H_ */
```

Lock.cpp

```
#include "Lock.h"
Lock::Lock() {}
Lock::~Lock() {}
```

DigitalLock.h

```
#ifndef DIGITALLOCK_H_
#define DIGITALLOCK_H_
#include <iostream>
using std::cout;
#include "User.h"
#include "Lock.h"

//Point 3 : Inherited abstract Lock class in Implemented the
unimplemented pure virtual functions.
class DigitalLock: public Lock {
public:
	DigitalLock();
	virtual ~DigitalLock();
	virtual void openLock(User user);

	static string positiveLight;
	static string negativeLight;
	static void informPolice();
	void printLockType();

protected:
	void unlockFeedback();
	bool lockState = false;
	virtual void setLockState(bool lockState);
	virtual bool getLockState();
	//	overriden getLockType
		virtual string getLockType();

private:
	string lockType = "Digital Lock";
	User owner = getOwnerData();
	void init();

	virtual void digitalLockType(string lockType);
	virtual bool autenticateKeyData(int keyData);
	virtual void startScreenDisplay();
	virtual void runDigitalScheme();
	virtual User getOwnerData();
	virtual void displayLockName();
	virtual void soundBeep();

};

#endif /* DIGITALLOCK_H_ */
```

DigitalLock.cpp

```
#include "DigitalLock.h"

//Point 16 Static states and methods
string DigitalLock::positiveLight = "Green";
string DigitalLock::negativeLight = "Red";
```

```cpp
//Point 16 Static states and methods
void DigitalLock::informPolice() {
	cout << DigitalLock::negativeLight << "SOS Intruder" << endl;
}

DigitalLock::DigitalLock() :
		Lock() {
	init();
}

void DigitalLock::init() {
	runDigitalScheme();
	startScreenDisplay();
	displayLockName();
}

DigitalLock::~DigitalLock() {

}

void DigitalLock::openLock(User user) {
	cout << "Digital lock: User name is " + user.name << endl;
	cout << "DigitalLock  is now " << (getLockState() ? "opened" :
"closed")
			<< endl;

	if (!getLockState()) {
		if (autenticateKeyData(user.pin)) {
			setLockState(true);
			unlockFeedback();
			cout << "Digital Lock " << (getLockState() ? "opened" :
"closed")
					<< endl;
			setLockState(false);
			cout << "User: " << user.name
					<< (getLockState() ? " opened" : " closed")
<< " the lock "
					<< endl;
		} else {
			cout << "DigitalLock access failed" << endl;
			cout << "Digital Lock remain"
					<< (getLockState() ? "opened" : "closed")
<< endl;
			informPolice();
		}
	}
}

void DigitalLock::digitalLockType(string lockType) {
	cout << "lockType" + lockType << endl;
}

void DigitalLock::unlockFeedback() {
	soundBeep();
}

bool DigitalLock::getLockState() {
	return DigitalLock::lockState;
}
```

```
// Point 10
string DigitalLock::getLockType() {
	return DigitalLock::lockType;
}

// Point 10
void DigitalLock::printLockType() {
	cout << " Lock type: " << getLockType() << endl;
}


User DigitalLock::getOwnerData() {

	User owner("vishnu", 1234);
	return owner;
}

bool DigitalLock::autenticateKeyData(int pinKeyData) {
	if (pinKeyData == owner.pin) {
		cout << "DigitalLock: User Data is " << pinKeyData << endl;
		return true;
	}
	return false;
}

void DigitalLock::runDigitalScheme() {
	cout << "DigitalLock: Running digital lock schema" << endl;
}

void DigitalLock::startScreenDisplay() {
	cout << "DigitalLock: Showing startScreenDisplay" << endl;
}

void DigitalLock::soundBeep() {
	cout << "Sound:: Lock Beeps..." << endl;
}

void DigitalLock::displayLockName() {
	cout << "DigitalLock is ready to provide access" << endl;
}

void DigitalLock::setLockState(bool lockState) {
	DigitalLock::lockState = lockState;
}
```

KeyPadLock.h

```
#ifndef KEYPADLOCK_H_
#define KEYPADLOCK_H_
#include "User.h"
#include "DigitalLock.h"
#include <iostream>
using std::cout;
using std::endl;

class KeyPadLock: public DigitalLock {




public:
	KeyPadLock();
	virtual ~KeyPadLock();
	virtual void openLock(User user);
	bool getLockState();
//Point 10 Dynamic binding
//getLockType() gets called based on the object type when its virtual.
otherwise it calls the same method in parent class;
	virtual string getLockType();

protected:
	virtual void setLockState(bool lockState);

private:
	string lockType = "KeyPadLock";
	User owner = getOwnerData();
	bool lockState = false;

	virtual bool autenticateKeyData(int pinKeyData);
	virtual User getOwnerData();
};

#endif /* KEYPADLOCK_H_ */
```

KeypadLock.cpp

```
#include "KeyPadLock.h"

KeyPadLock::KeyPadLock() {
}

KeyPadLock::~KeyPadLock() {
}
```

```
void KeyPadLock::openLock(User user) {
	cout << "KeyPadLock: User is " + user.name << endl;
	cout << "KeyPadLock  is now " << (getLockState() ? "opened" : "closed")
			<< endl;
	if (!getLockState()) {
		if (autenticateKeyData(user.pin)) {
			unlockFeedback();
			setLockState(true);
cout << "KeyPadLock  is " << (getLockState() ? "opened" :"closed")<<endl;
			setLockState(false);
cout << "User: " << user.name<< (getLockState() ? " opened" : " closed")<< "
the lock "<< endl;
		} else {
cout << "KeyPadLock: User->" + user.name + " Failed to authenticate"<<endl;
			cout << "KeyPadLock remain "<< (getLockState() ? "opened"
:"closed") << endl;
		}
	} else {
		cout << "KeyPadLock is already "
				<< (getLockState() ? "opened" : "closed") << endl;
	}
}

// Point 9
string KeyPadLock::getLockType() {
	return KeyPadLock::lockType;
}

bool KeyPadLock::getLockState() {
	return KeyPadLock::lockState;
}
void KeyPadLock::setLockState(bool lockState) {
	KeyPadLock::lockState = lockState;
}

bool KeyPadLock::autenticateKeyData(int pinKeyData) {
	if (pinKeyData == owner.pin) {
		return true;
	}
	return false;
}

User KeyPadLock::getOwnerData() {
	User owner("vishnu", 1234);
	return owner;
}
```

FingerPrintLock.h

```
#ifndef FINGERPRINTLOCK_H_
#define FINGERPRINTLOCK_H_
#include <iostream>
using std::cout;
using std::endl;
#include "User.h"
#include "DigitalLock.h"
```

```
class FingerPrintLock: public DigitalLock {
public:
	FingerPrintLock();
	virtual ~FingerPrintLock();
	virtual void openLock(User user);

private:
	string fingerPrintInput;
	User owner = getOwnerData();

	bool lockState = false;
	virtual bool getLockState();
	virtual void setLockState(bool lockState);

	virtual bool autenticateKeyData(string fingerPrint);
	virtual User getOwnerData();
};

#endif /* FINGERPRINTLOCK_H_ */
```

FingerPrintLock.cpp

```
#include "FingerPrintLock.h"
#include "KeyPadLock.h"

FingerPrintLock::FingerPrintLock() {
}

FingerPrintLock::~FingerPrintLock() {
}

void FingerPrintLock::openLock(User user) {
	cout << "FingerPrintLock: User is " + user.name << endl;
	cout << "FingerPrintLock  is now " << (getLockState() ? "opened" :
"closed")
		<< endl;

	if (!getLockState()) {
		if (autenticateKeyData(user.fingerPrint)) {
			setLockState(true);
			unlockFeedback();
			cout << "FingerPrintLock  is "<< (getLockState() ?
"opened" : "closed") << endl;
			cout << "Shows" << positiveLight << " Light";
			setLockState(false);
			cout << "User: " << user.name<< (getLockState() ? "
opened" : " closed") << " the lock "
				<< endl;
		} else {
			cout<< "FingerPrintLock: User->" + user.name+ " Failed
to authenticate" << endl;
			unlockFeedback();
			setLockState(true);
			cout << "FingerPrintLock remain "<< (getLockState() ?
"opened" : "closed") << endl;
		}
	} else {
```

```
            cout << "FingerPrintLock is already "<< (getLockState() ?
"opened" : "closed") << endl;
      }
}

bool FingerPrintLock::getLockState() {
      return FingerPrintLock::lockState;
}

bool FingerPrintLock::autenticateKeyData(string fingerPrint) {
      return (fingerPrint == owner.fingerPrint) ? true : false;;
}

void FingerPrintLock::setLockState(bool lockState) {
      FingerPrintLock::lockState = lockState;
}

User FingerPrintLock::getOwnerData() {
      User owner("vishnu", "fingerprint");

      return owner;
}
```

Hybridlock.h

```
#ifndef HYBRIDLOCK_H_
#define HYBRIDLOCK_H_
#include <iostream>
#include "User.h"
#include "FingerPrintLock.h"
#include "KeyPadLock.h"
using std::cout;

//Point 1: Multiple Inheritance: inherited both FingerPrintLock &
KeyPadLock
class HybridLock: public FingerPrintLock, KeyPadLock {
public:
      int securityLevel;
      string lockOS;
      HybridLock(string lockOS, int securityLevel);
//    HybridLock(HybridLock hybridlock);
      //Point
      //Point 14
      //modified copy constructor
      HybridLock(const HybridLock &inputHybridLock);
      virtual ~HybridLock();

      //operator overloading : over loaded == > and + operators
      bool operator ==(HybridLock);
      bool operator >(HybridLock hybridLock);
      HybridLock operator +(HybridLock);
      HybridLock& operator =(const HybridLock &hybridlock);
      virtual void openLock(User user);
```

```
protected:
	virtual void setLockOs(string lockOS);
	virtual void setSecurityLevel(int securityLevel);
	virtual void setCustomTheme(string customTheme);

private:
	string lockType = "HybridLock";
	string fingerPrintInput;
	string customTheme;
	User owner = getOwnerData();

	bool lockState = false;
	virtual bool getLockState();
	virtual void setLockState(bool lockState);
	virtual bool autenticateKeyData(string fingerPrintkeyData);
	virtual bool autenticateKeyData(int pinKeyData);
	User getOwnerData();
};

#endif /* HYBRIDLOCK_H_ */
```

Hybridlock.cpp

```
#include "HybridLock.h"
#include "User.h"
#include <iostream>
using std::cout;

HybridLock::HybridLock(string lockOS, int securityLevel) {
	cout << "H L constructor" << endl;
	this->lockOS = lockOS;
	this->securityLevel = securityLevel;
}

//Point 14 Modified copy constructor
HybridLock::HybridLock(const HybridLock &inputHybridLock) {
	cout << "H L COPY constructor" << endl;
	this->lockOS = inputHybridLock.lockOS;
//	security level fall by one when someone and copy this Lock
	this->securityLevel = inputHybridLock.securityLevel - 1;
}

//Point 12 operator overloading
HybridLock& HybridLock::operator =(const HybridLock &hybridLock) {
	cout << "H L  = constructor" << endl;
	if (this != &hybridLock) {
		lockOS = hybridLock.lockOS;
		securityLevel = hybridLock.securityLevel;
	}
	return *this;
}
//Point 11 operator overloading
bool HybridLock::operator ==(HybridLock hybridLock) {
	if (hybridLock.securityLevel == securityLevel
			&& hybridLock.lockOS == lockOS) {
		return true;
	} else {
		return false;
	}
}
```

```
//Point 11 operator overloading
HybridLock HybridLock::operator +(HybridLock hybridLock) {
	return HybridLock(lockOS, securityLevel + hybridLock.securityLevel);
}
//Point 11 operator overloading
bool HybridLock::operator >(HybridLock hybridLock) {
	if (securityLevel > hybridLock.securityLevel) {
		return true;
	} else {
		return false;
	}
}

HybridLock::~HybridLock() {
}

void HybridLock::setLockOs(string lockOS) {
	this->lockOS = lockOS;
}
void HybridLock::setSecurityLevel(int securityLevel) {
	this->securityLevel = securityLevel;
}

void HybridLock::openLock(User user) {
	cout << "HybridLock user name is: " + user.name << endl;
	cout << "HybridLock  is now " << (getLockState() ? "opened" : "closed")
			<< endl;
	if (!getLockState()) {
		if (user.pin != 0 || user.fingerPrint != "") {
			if (autenticateKeyData(user.fingerPrint) == true
					&& autenticateKeyData(user.pin) == true
					&& user.name == "vishnu") {
				setLockState(true);
				FingerPrintLock::unlockFeedback();
cout << "HybridLock  is "<< (getLockState() ? "opened" : "closed") << endl;
cout << "Shows" << positiveLight << " Light";
} else {
cout << "HybridLock: Access Failed for the User: " + user.name<< endl;
cout << "HybridLock remain "<< (getLockState() ? "opened" : "closed") << endl;
cout << "Shows" << this->lockOS << " Light";
informPolice();
			}
		}
	}
}

void HybridLock::setCustomTheme(string customTheme) {
	this->customTheme = customTheme;
	cout << "Hybrid Lock: customTheme is " << customTheme << endl;
}


bool HybridLock::getLockState() {
	return HybridLock::lockState;
}

void HybridLock::setLockState(bool lockState) {
	HybridLock::lockState = lockState;
}
```

```cpp
// function overloading
bool HybridLock::autenticateKeyData(string fingerPrintkeyData) {
	if (fingerPrintkeyData == owner.fingerPrint) {
		cout << "fingerPrint Data matched" << endl;
		return true;
	}
	return false;
}

bool HybridLock::autenticateKeyData(int pinKeyData) {
	if (pinKeyData == owner.pin) {
		cout << "Hybrid Lock: User pin Data matched" << endl;
		return true;
	}
	return false;
}

User HybridLock::getOwnerData() {
	User owner("vishnu", 1234, "fingerprint");
	return owner;
}
```

User.h

```cpp
#ifndef USER_H_
#define USER_H_
#include <iostream>
using std::cout;
using std::endl;
using std::string;

class User {
public:
	string name;
	string password;
	int pin = 0;
	string fingerPrint;

	virtual ~User();
	User(string name);
	User(string name, int pin, string fingerPrint);
	User(string name, int pin);
	User(string name, string fingerPrint);
	void printUserDetails();
	void operator =(const User &user);
};

#endif /* USER_H_ */
```

User.cpp

```cpp
#include "User.h"

void markAsVulnerable(User &user){
	user.name = "vulnerable user";
}

User::User(string name) {
	this->name = name;
	// TODO Auto-generated constructor stub
}

//Constructor overloading
User::User(string name, int pin, string fingerPrint) {
	this->name = name;
	this->pin = pin;
	this->fingerPrint = fingerPrint;
}

User::User(string name, int pin) {
	this->name = name;
	this->pin = pin;
}

User::User(string name, string fingerPrint) {
	this->name = name;
	this->fingerPrint = fingerPrint;
}

User::~User() {
//	Point 9
cout << "User destructor called " << endl;
// delete ptr; delete any pointers in destructor to deallocate
dynamically allocated memory
}

// operator overloading permitted only for hybrid lock
void User::operator =(const User &user) {
	this->name = name;
	this->pin = pin;
	this->fingerPrint = fingerPrint;
}

// Point 13 object passing itself to a function.
void User::printUserDetails() {
	cout << this->name << endl;
	markAsVulnerable(*this);
	cout << this->name << endl;
}
```