CS 247 S2011 Midterm

1. A)

```
class LicensePlate {
        static std::set<std::string> usedValues;
        static std::string generateValue();
        static bool isValid();

        std::string value;
        LicensePlate(LicensePlate&);
        LicensePlate& operator=(LicensePlate&);
public:
        LicensePlate();
        LicensePlate(std::string);
        LicensePlate(LicensePlate&&);
        LicensePlate& operator=(LicensePlate&&);
        friend std::istream& operator>> (std::istream&, LicensePlate&);
        friend std::istream& operator<< (std::ostream&, const LicensePlate&);
};
```

B) The default initial value is a randomly generated unique license plate number, since ensures license plates are unique and ensures no empty (invalid) license plates.

C) The license plate value is immutable, since license plates should not be able to be changed.

D) generateValue and isValid are member functions, since they are specific to this class and will be called from the class. The input and output stream operators are not member functions, since the left-hand operator is implicitly *this, and it would not make sense to call << or >> from LicensePlate instead of std.

Bonus) There is a static set variable that keeps track of used variables, and the copy constructor and copy assignment operator are private to prevent copying.

**note: if not using STL, you can create variables charOne, charTwo…charEight, curLength, and nextUpdateChar, and generate the LicensePlate values by incrementing appropriately.

2.

```
WatCard::WatCard(const StudentRecord* sr) : sr_(sr), barcode_(newBarcode()),
PIN_(nullptr), active(false), expiry(Date()), accounts_(new Account[NUM]) {
        expiry.incYears(2);
        accounts_[FLEX] = new FlexDollars();
}
```

3. A) Default constructor: The generated default constructor would not be appropriate, since it would not properly initialize a unique barcode, accounts, or expiry date. Destructor: The generated destructor would not be appropriate, since it would not

properly free memory from the PIN and accounts.
Copy constructor: This would not be appropriate, since it would not deep copy values correctly (and it also doesn't make sense to copy a WatCard?)
Assignment operator: This would not be appropriate, since it would not deep copy values correctly (and it also doesn't make sense to copy a WatCard?)

B)
```
MealPlanFunds& MealPlanFunds::operator= (MealPlanFunds& other) {
    MealPlanFunds temp {other};
    using std::swap;
    swap(plan, temp.plan);
}
```

4.

A)
```
class Account {
    Money balance;
    std::vector<Transaction*> transactionHistory;
protected:
    Money& balance() const;
    void balanceIs(Money&);
public:
    Account();
    Account(Money&);
    ~Account() = 0;
    bool purchase(const Money& amount);
    void transactionHistory(ostream&, Date, Date) const;
    virtual void print(ostream&) const;
};
```

B)
```
bool Account::purchase(Money& amount) {
    if (balance >= amount) {
        balance = balance - amount;
        transactions.push_back(new Transaction(this, Date(), "debit", amount);
        return true;
    } else {
        return false;
    }
}
```

5.

A) `MealPlanFunds::MealPlanFunds(MealPlan* plan) : plan(plan) {}`

B)
```
bool MealPlanFunds::purchase(Money& amount) {
    Money discounted = amount*discount();
    if (balance() >= discounted) {
        balanceIs(balance() - discounted);
        transactions.push_back(new Transaction(this, Date(), "debit", discounted);
        return true;
    } else {
        return false;
    }
```

```
}

C)
void MealPlanFunds::endMealPlan(TransferMealPlanFunds& toAccount) {
	plan = nullptr;
	toAccount.balanceIs(toAccount.balance() + balance());
}
```

6.

```
A)
bool WatCard::purchase(AccountType accountType, const Money& amount) {
	if (accountType == MEAL || accountType == TRANSFER) {
		if (accounts_[accountType] && accounts_[accountType]->purchase(amount)) {
			return true;
		}
	}
	return accounts_[FLEX]->purchase(amount);
}

B)
void addDollars(Money& amount) {
	accounts_[FLEX]->deposit(amount);
}

C)
void WatCard::transactionHistory(std::ostream &sout, const Date& start, const Date& end) {
	accounts_[FLEX]->transactionHistory(sout, start, end);
	if (accounts_[MEAL]) {
		accounts_[MEAL]->transactionHistory(sout, start, end);
	}
	if (accounts_[TRANSFER]) {
		accounts_[TRANSFER]->transactionHistory(sout, start, end);
	}
}
```

7.

A) The Liskov Substitutability Principle states that pointers to a Base class must be blindly substitutable by Child class objects. Since TransferMealPlanFunds does not have some of the functions of MealPlanFunds, like endMealPlan and discount(), it should not be a subclass of MealPlanFunds.

B) The Law of Demeter is the principle of least knowledge - each unit should have limited knowledge about other units and should only talk to immediate friends. The WatCard composite class adheres to this principle - WatCard only accesses Account, Account only accesses its immediate child classes and Transaction, and only MealPlanFunds and TransferMealPlanFunds accesses MealPlan.

8.

A) Not declaring o to be constant means the function could be written to accidentally modify o when it shouldn't be modified.

B) The implicit this parameter is the left-hand side of the operator.

C)

```
struct HourImpl {
	int hour;
};
class Hour {
	HourImpl *pImpl;
	public:
		Hour(int);
		int hour() const;
};
```

D)

```
try {
	assert(pImpl->hour >= 20 && pImpl->hour <= 23);
} catch(…) {
	cout << "Hour must be between 0 and 23" << endl;
}
```

E) Converting the illegal value into a legal value changes the state of the data in an unspecified way.

F)

```
#ifndef A_H
#define A_H

class B; // Forward declaration
…
```

G)

```
B.o : B.cpp B.h A.h
\t g++ -std=c++14 -c B.cpp
```

Other G) I don't think we need to know this?

H)