1a. (Answered by Mike)

The ChildAccount class would not be able to inherit canWithdraw from the BankAccount class, because ChildAccount inherits minimumBalance from SavingsAccount, and this would cause issues as you can withdraw from the BankAccount class so long as your balance is greater than zero, whereas the ChildAccount’s balance would needed to be higher than its minimum balance at all times.

To solve the issue mentioned above, you could make the BankAccount::canWithdraw() method virtual to allow it to be overridden, then you would be able to specify different functionality for the method.

1b. (Answered by Mike)

Using a BankAccount instance instead of a pointer in this case study would not be appropriate, because the type of account is decided on the fly and so the object is created dynamically and stored in the BankAccount pointer. When I say the type of account is decided on the fly, I mean there is a hierarchy of BankAccount types and we do not know which type we will be using at compile-time; this is decided at run-time.

The type of account is taken from the file name which stores the corresponding account information. It is the first digit of the account number in the file name that is used; the corresponding types are ‘0’ for BankAccount, ‘1’ for CurrentAccount, ‘2’ for SavingsAccount, ‘3’ for ChildAccount and ‘4’ for ISAAccount. BankAccount and SavingsAccount types will not explicitly be used further in the development process, because these two classes will become abstract and you cannot instantiate an abstract class. An abstract class cannot be instantiated, because they contain at least one pure virtual method whose function body is only implemented by the derived class.

A specific example to illustrate my answer of the type being decided dynamically, would be account 101 with sort code 00-44; the first digit is a ‘1’, so the user would enter these account details, the application will take that digit out of the file name and use it in a switch statement which will then execute the following line p\_BA = new CurrentAccount; (remember above we said ‘1’ relates to the type CurrentAccount). Now the BankAccount pointer points to a CurrentAccount object (which is a BankAccount as CurrentAccount is derived from BankAccount), so when the next line is executed p\_BA->readInBankAccountFromFile( aBAFileName );, it will call the overloaded operator ‘>>’, which will then choose the corresponding version of getDataFromStream to the CurrentAccount class.

When I mentioned corresponding version of getDataFromStream to the CurrentAccount class, the method is pure virtual inside BankAccount and is implemented by any non-abstract, derived method and the version of that method is decided dynamically, based on the type our pointer points to.

Traversing back up this answer to where we read in ‘1’ for CurrentAccount and made our pointer point to this new instance; if the value we read in were a ‘3’, then instead of executing p\_BA = new CurrentAccount; we would have executed p\_BA = new ChildAccount; and the getDataFromStream that corresponds to the ChildAccount class would have been called instead.

The reason getDataFromStream needs to be overridden is, because each account type’s instance types and number of instances vary.

1c. (Answered by Mike)

The nature of the relationship between CashCard and List<string> is that of an aggregation relationship, because ListT is a “part of” CashCard and ListT can exist independent of the parent. The relationship is an Association type, and the reason the nature is that of aggregation and not dependency or composition is, because composition means the object is entirely composed of the other object and in our case, there is another component to CashCard; it is not a dependency relationship, because dependency means uses temporarily (meaning it is used as a return type or parameter to a member function, and is not used by the class in any other way – these are the only C++ mechanisms it is involved with in its implementation).

1d. (Answered by Mike)

No, the class UserInterface is not an abstract class. An abstract class must have at least one pure virtual function declaration, which UserInterface does not and if it was an abstract class, then the class should have been designed to specifically be a base class for other classes to derive from.

The UserInterface class should not be an abstract class, because it is fit-for-purpose in its current implementation. If UserInterface was to be made an abstract class, then you would not be able to directly create an instance for use as the pure virtual functions would require implementation, and if they were implemented inside the UserInterface class, then the class would no longer be abstract.

1e. (Answered by Mike)

Date::currentDate() is declared as static, because it’s implementation is not object specific; it does not rely on an instances’ data members to function correctly, and its function is the same throughout each instance of the ‘Date’ class.

Also, making the method static, makes it more openly available, because you are not required to instantiate Date to use this method; it can be called direct from inside or outside of the class. Although instances can still call this method using the dot/pointer syntax, this method can be called direct like so, Date::currentDate();

With the method not specifically belonging to an instance, it cannot call instantiated Date members using the ‘this’ pointer, because there is no way to tell which instance you are referring to. Although the currentDate method does not specifically belong to an instance, the method itself can still be called using the ‘this’ pointer or dot/pointer notation, because the method still belongs to the class.

1f. (Answered by Mike)

An issue with the given code is that t is declared multiple times, but assuming each line is to be implemented alone:-

**Line 1**

Here is a list of what would be called in order for line 1:

* The Time::Time( int h, int m, int s) basic constructor of a new Time instance
* Overloaded operator + method of t1
* The conversion constructor of t
* The destructor of the new Time instance

The destructor of t followed by the destructor of t1 would be called later. t1’s constructor will have been called prior to line 1.

Line 1 should compile without making any changes and work without any problems.

**Line 2**

Here is a list of what would be called in order for line 2:

* The Time::Time( long) conversion constructor of a new Time instance
* Overloaded operator + method of the new Time instance
* The conversion constructor of t
* The destructor of the new Time instance

The destructor of t followed by the destructor of t1 would be called later. t1’s constructor will have been called prior to line 1.

Line 2 should compile without making any changes and work without any problems.

**Line 3**

Here is a list of what would be called in order for line 3:

* Conversion constructor of a new Time instance
* Overloaded operator + method of t1
* The conversion constructor of t
* The destructor of the new Time instance

The destructor of t followed by the destructor of t1 would be called later. t1’s constructor will have been called prior to line 3.

Line 3 should compile without making any changes and work without any problems.

**Line 4**

Line 4 would not compile if used, because the operator + cannot add a number literal to a Time instance by default. To fix the problem you could overload the + operator:

const long operator +( const long num, const Time &t )

{

return num + t.getSeconds();

}

Another way to do this, would be:

const Time operator +( const long num, const Time &t )

{

return Time( num + t.getSeconds() );

}

There are a few ways this can be done, causing different constructors to be called.

Now we have fixed the issue, if we were to use the first solution we came up with, here is a list of what would be called in order for line 4:

* The overloaded method we created above
* Time::getSeconds() method (called by our overloaded method)
* The conversion constructor of t

The destructor of t followed by the destructor of t1 would be called later. t1’s constructor will have been called prior to line 4.

All four lines will now produce the same result as each other.

Objects are created and destroyed in a LIFO (Last in first out) sequence; this means that the last object created, will be the first to be destroyed and the first object created will be the last to be destroyed.

1g. (Answered by Mike)

No, the version given is not correct. Although assert statements work in debug mode, it will not work in release mode and is open to run-time errors. There is a missing bracket on this line of execution Transaction firstTr(newestTransaction();. In terms of what the code is expected to do, the answer to this being correct is still no, because olderTransactions only makes a copy of the current instance, and the resulting list is at no point assigned back to its instance.

The code in itself is not as optimized as these few lines can be; there are two extra calls to newestTransaction and olderTransactions than was required.

Here is my working version:

void TransactionList::deleteGivenTransaction( const Transaction& tr )

{

if( !size() )

return;

Transaction firstTr( newestTransaction() );

\*this = olderTransactions();

if( firstTr == tr )

return;

deleteGivenTransaction( tr );

addNewTransaction( firstTr );

}

Whilst answering this question, I am assuming all other necessary functionality is in place i.e. the overloaded comparison operator ==.

To start with I removed the assert statement and replaced it with a check on the size, if the size is 0 or lower, we would then return without executing this function and we would prevent any run-time errors. Next I am storing the first transaction in the variable instead of inside the else part of the conditional statement, this enabled me to use the transaction variable in the equality check and refer to the transaction later using the variable, just in one call to get the transaction.

Also, in the interest of keeping our code optimised, I moved the assignment of the instances tail to itself (in effect removing the first transaction); we are able to do this, because it is required in both blocks of the original statements. If the first transaction is the one we’re looking for, we just simply return as the transaction is no longer a part of this instance, but if it is not, then we have already saved the first element to re-add on the last line of this function.

Next, I have removed the else part of the conditional statement, because these statements will not be reached if the above condition is met. We now use recursion to continue our quest to find the transaction passed in, or until we have walked the entire list and not found it. On the final line of code, each transaction removed in the recursion process is added back in their initial order.

1h. (Answered by Mike)

If I were to make BankAccount::prepareFormattedAccountDetails as virtual, I would then be able to override this method in derived classes.

I would want to do this, because each type of account has different details; an example would be: SavingsAccount has a member instance called minimumBalance\_, which represents the lowest balance an instance of type SavingsAccount can have and BankAccount where the function is initially declared does not have this.

Although BankAccount and SavingsAccount are to be made abstract classes, if they were not then the functionality would be different in the sense that BankAccount does not have a minimumBalance\_ to be formatted, whereas SavingsAccount does.

1i. (Answered by Mike)

No, the function call would not be valid, because the compiler would just see the pointer p\_theActiveAccount as a pointer to a BankAccount instance, which does not have the given method.

In order to solve our problem, we would need to create a CurrentAccount pointer and make it point to our bank account instance currently store in p\_theActiveAccount. We cannot directly do this, so we would need to downcast p\_theActiveAccount to tell the compiler (in pseudo terms), “This is supposed to be a CurrentAccount pointer”, then we would make the call from our new pointer.

We would do this like so:

CurrentAccount \*currentAccount = (CurrentAccount \*)p\_theActiveAccount\_;

currentAccount->getOverdraftLimit();

1j (Answered by Mike)

No, the method CashPoint::m1\_produceBalance() could not be written as given in the question, because BankAccount::balance\_ is a private member variable and cannot be accessed direct. You could move the data member to the public scope within the BankAccount class, but this would create a security flaw and would go against OOP principles. A more ideal alternative would be to use the code below:

void CashPoint::m1\_produceBalance() const

{

theUI\_.showProduceBalanceOnScreen(

p\_theActiveAccount\_->getBalance() );

}

This would still take away the need for the temporary double variable, and it would still produce the same functionality. The difference is, my change makes use of the accessor method BankAccount::getBalance() instead of accessing the member variable BankAccount::balance\_ direct.

For clarification on scope within classes, please read below.

public - anyone can see, you can access methods and member variables placed here outside of the class; whether it's from main or some random method.  
  
protected - this is semi-private; you cannot access the members placed here outside of the class i.e. in main or some random method, but when there is a derived class, then it will inherit these members also.  
  
private - only this class’ member functions can access what is in here, and if access to the values is required elsewhere, then they will have to use one of the access methods I have placed in public (or protected if it's only an inherited class that needs access).