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.

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).