1. Why is *protected* scope necessary?

A data member or member function declared in a protected section of a class is only accessed by member functions and friends of that class, and by member functions and friends of derived classes (if the member were in private scope then it would only be accessed by members and friends of that class, not the derived class).

If you want to limit subclass access to your base class member variables, then private scope is the way to go; if subclasses are allowed to enter the internal state, then protected scope suffices. Protected scope naturally creates a coupling between derived classes and their bases. You must also consider the fact that any derived class could modify the variable of your base class, so there is less enforcement and possibly the chance of really screwing up chains of derived classes by modifying such base class values without an understanding of the relationships between base and derived classes (assuming one has access to this info). You are still given the flexibility of removing or changing protected scope to private scope in such cases with minor modification on your end.

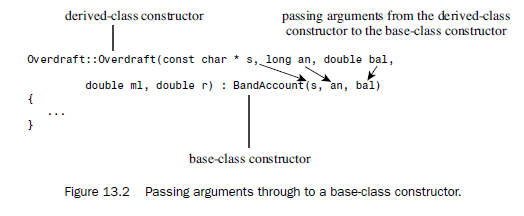
1. Discuss the wisdom, from a software engineering perspective, of using protected members.

If some developer, say Chad, comes along then decides to create a derived class from your base class using protected scope, then you sure hope he knows how the base class works as he may modify the state of variables used in other inherited classes. If instead you took the private scope route, Chad will only have access to the public interface and upon releasing his derived class, you are then free to change/remove the protected members from the base class without damage to his class. You could, of course, want to restrict the coupling between your base class and derived classes from future Chad subclasses and can opt for the private scope as well.

If hundreds of Chads outside your team will create hundreds of derived classes from your class then it would be wise to create a protected interface using private data. If you keep design within your own reasonably connected team with adequate oversight, protected members may simplify writing the code as the implications are known. You could also essentially create an interface for one set of users/customers and a protected interface for your disciples/team. If you generally want to ensure that derived classes from your masterful base class don’t break should you opt to change its data structure or the like, then privatizing data is the way to go.

1. When constructing an object of a derived type, is a base class constructor always invoked? Explain.

A derived class needs its own constructors as it can add additional data members and member functions if needed. The constructors have to provide data for inherited members *and* any new members*.* Also, the derived class has to work through public base-class methods to access private members of the base class, so that derived-class constructors automatically have to use the base-class constructor. Here is a helpful figure from the text:



When a program constructs a subclass object, it will first construct the base-class object BUT creating the derived object invokes the derived-class constructor, not a base-class constructor. That means the base-class object would be created before entering the body of the derived class constructor (on the derived class header using list initialization). Even if you opt to omit calling the base-class constructor (not using list initialization), the default base-class constructor is used. Unless you intend for the default constructor be used, then you should explicitly code the correct base-class constructor call. So, the following achieves the same result as the program uses the base-class default constructor if not member initialized in the derived-class header:

Derived::Derived(double x, const string &newMember){…}

Derived::Derived(double x, const string &newMember) : Base(x){…}

In sum, the base-class object is constructed first (but the derived-class constructor is invoked when creating this subclass object); the derived-class constructor should pass base-class info to a base-class constructor via member initialization; and derived-class constructors should initialize the new data members. The derived-class constructors don’t inherit the base-class constructors so there isn’t a need to make them virtual either. Inheritance means a derived object can use a base-class method, but, for constructors, the object doesn’t exist until after the constructor has…constructed.

1. Discuss the veracity of the following statement: For an object of some derived type, the base class constructor is invoked first, then from the constructor the derived class is invoked.

See prior discussion.

1. Why doesn't each object that has a virtual function associated with it have its own V-table?

The vtbl holds addresses of the virtual functions declared for objects of that class. The object of a base class and a derived class each have their own pointer to a separate “array” of addresses (thus each table itself has its own address). If the subclass redefines a virtual function, the vtbl holds the address of the new function and replaces the original address in its vtbl “chain”. If the subclass keeps the virtual function as is, the vtbl keeps the address of the original version. If the subclass adds a new virtual function, it adds this address to the vtbl. In this way, the accounting is such that no matter how many virtual functions are declared, only the vtbl address member is added to the object and the table size its pointer points to is that which varies.