Lecture 18

Conceptual Questions:

- **Do ALL virtual functions need to be implemented in derived classes? Extend the statement.**

Derived classes do not have to implement all virtual functions themselves. They only need to implement the pure ones.1 That means the Derived class in the question is correct. It inherits the bar implementation from its ancestor class, Abstract. (This assumes that Abstract:: bar is implemented somewhere. The code in the question declares the method, but doesn't define it. And even then, only if the derived class is going to be instantiated. If a derived class is not instantiated directly, but only exists as a base class of more derived classes, then it's those classes that are responsible for having all their pure virtual methods implemented. The "middle" class in the hierarchy is allowed to leave some pure virtual methods unimplemented, just like the base class. If the "middle" class does implement a pure virtual method, then its descendants will inherit that implementation, so they don't have to re-implement it themselves.

Only the pure virtual methods have to be implemented in derived classes, but you still need a definition (and not

just a declaration) of the other virtual methods. If you don't supply one, the linker might very well complain. So, just putting {} after your optional virtual method gives you an empty default implementation:

```
class Abstract {
public:
    virtual void foo() = 0; // pure virtual must be overridden
    virtual void bar() {}   // virtual with empty default implementation
};

class Derived : Abstract {
public:
    virtual void foo();
};
```

- **What is the importance of *friend* function in C++ if it can access the private members of a class? Where is the data security here?**

fn(b) and fn(a,b) have the exact same sematic power.You use friend function when -for syntactic reason belonging to the problem you want to solve or describe- the second form has to be preferred (for example, if f is commutative respect to a and b, the first form gives to a a special role that -in fact- doesn't have), or when the "owning class" must not be the first argument.

Friends are nothing more than "members of more classes" or "member whose owner is explicit". If you understand this, please stop listening to anyone talks about "encapsulation breaking" because of friendship. Friend don't break any encapsulation for the very simple reason that a fucntion cannot be a friend of a clas without that class wants it. They don't break anything. Friendship is just the construct to make "capsules" larger than classes. The simpler example is considering a class A that have to output itself on a stream.

You need an ostrem& operator<<(ostream&, const A&): this is required by the ostream syntax, but you need to access A insides, with A not being the first member.

- **How do we access private data members of a class outside the program in C++ language?**

There is no way to access private members of a class outside a program. But you can access them outside the class (not program) from main() function. either you can use public member function to access them but, you must be aware of this method. But there is one more way to access private member without using any public member from class. You can directly use pointer to access private data member of a class from main(). You can call it a loophole of C++.

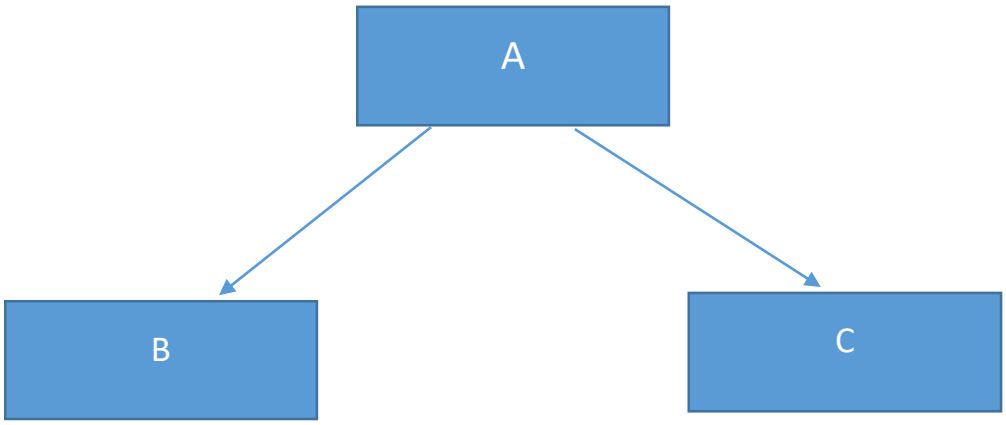
```
#include<iostream>
using namespace std;

class test
{
private:
    int x;
    int y;
public:
    test()
```

```
        {
         x = 0;
            y = 1;
        }
};

int main()
    test t;
    int* ptr = (int*)&t;

    cout<<(*ptr)<<endl;
    cout<< (*ptr+1)<<endl<<endl;

    *ptr = 10;
    *(ptr+1) = 12;

    cout<<(*ptr)<<endl;
    cout<< (*ptr+1)<<endl;
```

```
    return 0;
}
```

In above program we have assigned value to x and y using constructor. Then we create a pointer and typecasted the address of test object to integer and assign it to our pointer ptr. Then we can easily manipulate the value of x and y as shown in line 24 and 25. Here we are not using and public member function to access them, we are directly accessing them using pointer. In line 24 and 25 we are able to change its value and in line 27 28 we are displaying the private member using pointer.

- **Suppose there are three classes named "A", "B" and "C". Build a diagram to show how these classes are related. What do you need to do to hide the implementation details of the parent class obtained from the diagram?**



Abstraction is all about providing an additional layer with interfaces and abstract classes. This layer (interfaces and abstract classes) tells about what needs to be done, but NOT how. So hiding implementation is called as the abstraction. The best example to understand the concept of abstraction is that all J2EE/JMS specifications provide abstractions (typically interfaces) to the application vendors and then these interfaces will be implemented by the different vendors (like Tomcat/JBoss/Weblogic/ IBM/etc..) with the actual definitions/behavior (called implementations) for the specifications. Abstraction talks only about WHAT needs to be done and Implementation tells about HOW it should be done. Abstraction provides the power of injecting the behavior at Runtime (which is Polymorphism). Now, taking Spring framework (or infact any DI framework like Guice, etc..) taking as example, the Spring DI container injects the provided bean (through xml or annotations) implementation object (implementation) at runtime to the given interface type (abstraction).

Encapsulation is a related concept, not quite the same, but not entirely independent. The Wikipedia definition is quite bad unfortunately, because it focuses on the language. The concept of encapsulation means to protect details inside a unit/method/class/module/etc. It is relevant for both the

"outside" and "inside" of an object. This is the thing you don't have, if for a new "field" on some business object you have to modify multiple "layers" like persistence, business, gui, api, etc. I tend to agree with the Booch quote you referenced, but the problem with it is that it is quite misunderstood what "essential characteristics" of an object are. Encapsulation would for example never allow direct access to internal fields of an object. This means getters (and especially setters) are a no-no. One can argue about "degrees" of encapsulation of course, but most of the business code I've seen at least fails encapsulation quite spectacularly.