Programming Two

Lecture 3: A Look at the fundamental theories of Object Oriented Programming

Introduction
The aims of this lecture are to give you a good introduction to the different programming ideas of Object Oriented Programming (OOP).

Links
The Types of Programming: http://staff.ncst.ernet.in/1/1/sasi/paraintro.html
Inheritance in Classes: http://www.cplusplus.com/doc/tutorial/tut4-3.html

Code
Week 3 -> Inheritence Example

Types of Programming
During the course you have heard me mention object Oriented programming, but what exactly does that mean?
Well the types of programming paradigms that are defined are many: Functional, Imperative (or Procedural), Logical, Object-Oriented.
Most programming languages use a few of these paradigms. I am going to discuss the main two that arise, Object Oriented and procedural, and hopefully help define what Object Oriented programming actually means.

What is an object?
So we've mentioned aspects of OOP programming, but what exactly is an object.
Well this is actually quite tricky are there are many definitions of an object lying around. The classical definition is:

"An object has state, behaviour, and identity; the structure and behaviour of similar objects are defined in their common class; the terms instance and object are interchangeable" (Brooch '94)

For now it is enough to associate an instance of a class with object because – "A class is a general term denoting classification and also has a new meaning in object-oriented methods. Within the OO context, a class is a specification of structure (instance variables), behaviour (methods), and inheritance (parents, or recursive structure and behaviour) for objects". (objectfaq.com)

Classes/Objects are most often modelled around real world concepts such as chair or dog.
For instance a dog class could have the data weight, age and the member functions bark, wag tail etc.
The class has therefore been modelled around a real world concept.

OOP Paradigms

There are three or four main object oriented principles or paradigms which are important to an overall object oriented approach.
These principles are Abstraction, Encapsulation, Modularity and Inheritance.

Some people group Encapsulation and Abstraction together because they are quite similar. We will look at them separately and you can decide.

The exact definition of an OOP language varies, I've seen quite a few, generally a language that implements the four major elements as discussed below may be described as Object Oriented.

The four main principles

To examine the principal paradigms of OOP we will be using a tutorial, slightly modified, provided by GTS Learning (http://www.gtslearning.com/)

Abstraction

"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provides crisply defined conceptual boundaries, relative to the perspective of the viewer."
Grady Booch (Object Oriented Analysis and Design)

Essentially abstraction is the process of design which allows us to ignore details, commonly resulting in a top-down approach.

By concentrating on the essential characteristics of an object which distinguish it from all other objects allows implementation to be ignored, or abstracted. In other words abstraction allows questions such as ' What can it do? ' rather than ' How is it going to do this?'. The essence, then, is to define an object in terms of those features which are unique to it, or which simply summarise its nature or function.

To access these features and use the object C++ uses Public Member Functions, other languages talk of the PROTOCOL or INTERFACE. Something that we will come on to when discussing typing in C++ virtual functions and late binding, these are particularly good as the use of them allow the programmer to forget the details of data being operated on.
It is worth noting that generally all member data in the class should be abstracted from the user of the class. That means that to access and alter data we provide functions that enable the user (who is generally a programmer) access to it in the way we describe. All the user needs to know is what the function does and what parameters its returns and takes (if any).

The Example: There are many kinds of light bulb, but they all share the common function of providing a source of light. This view concentrates on the light bulb's essential function, whilst ignoring its implementation. Regardless of the nature of the light bulb - incandescent, fluorescent, or gas vapour - the main features are that you can turn it on and light is produced, or turn it off and the light ceases.

This could be shown with the following (incomplete) class:

```
class LightSource
{
    public:
        LightSource();
        ~LightSource();
        void TurnOn (void);
        void TurnOff (void);
        int IsOn (void);}
};
```

The class LightSource declares an abstraction of a light source, declaring the operators which may be performed on classes derived from it, namely turning the light on and off and determining its current state. At this stage the mechanisms for doing this are ignored.

Encapsulation
Encapsulation is sometimes called Information Hiding.
Encapsulation is complementary to abstraction and is used to describe the process of defining the code and data which form an object. (Many programmers erroneously interpret this to mean data hiding. It isn't: we hide both data and code).
Within the object, some data and code may be accessible directly as the interface of the object, while other parts of the object remain private. In fact it is usual for all data to be private, access to values being through the return of (public) member functions.

Java libraries and the C++ standard template libraries are good examples of this. They are classes that provide a lot of functionality we use these as a programmer without knowing the underlying code or being able to access the data directly.
Generally, therefore, it is a goal of encapsulation to ensure that all information and processes associated with an object are contained within its definition. It should not be possible to affect the state of an object other than through its public interface.

We can now expand the declaration (and definition: we will use in-line functions) of the LightSource class:

```
class LightSource
{
private:
    int lightState;

    void Light(void) = 0;
    void Dark (void) = 0;

public:
    LightSource() {lightState = 0;}
```

```
    ~LightSource();
    void TurnOn (void) { if(!lightState) Light(); lightState = 1;}
    void TurnOff (void) { if(lightState) Dark(); lightState = 0;}

int IsOn(void) {return (lightState != 0);}
};
```

Encapsulation has occurred:
The state of the light source is hidden from clients via the private variable lightState and may only be read or altered via the published interface
All of the operations which may be performed upon a light source are defined by the three public functions TurnOn, TurnOff and IsOn

It is fair to say that encapsulation can lead to complexity: it is not in itself a guarantee of simplicity and due care must be exercised.

Modularity
Modularity deals with the physical make up of the code, mainly its separation into code files or libraries.
Many languages, and C++ is no exception, allow the separate compilation of modules which are then linked together to form the executable code. The basic conventions of C still apply, but may be a little more formally expressed: Header (.h) files are effectively the interfaces of the different modules, and the .c or .cpp files contain the implementation. As to what should constitute a module, much depends on physical constraints.
It is generally considered good practice to have one class per header file and its members function defined in a complimentary source file.

In many senses modularity almost comes for free. A general rule is to group logically related classes and objects in the same module, setting up interfaces only to those parts other modules must see. Remember that the goal is twofold: the simplifying of documentation under the divide and rule principle, and the elimination of unnecessary compilation. The two ideals are COHESION, that is, groups of logically related abstractions; and LOOSE COUPLING, that is, minimal dependencies between modules.

Performing these operations will reduce the chance of unnecessary complexities and dependencies that make code much harder to follow and debug.

Once, then, the key abstractions have been identified, it is a relatively simple step to divide physically the implementation into coherent modules. Abstraction combined with encapsulation ensures that a given module will include all relevant functions and data. Encapsulation ensures that the modules are unaware of, and hence unaffected by, changes to the implementation details in other modules. This is a major advantage. Regardless of good intentions, in any large system programmers are often led to make coding decisions which depend on the internal implementation of another module; perhaps even relying on side-effects.

Modularity may of course be affected by other needs: separate processors in a multi-processor system; segment size limits; dynamic calling behaviour in virtual memory systems (consequent on the need for late binding); the building of libraries where reusability is the main objective; and work allocation in large project teams. In any event it is important to realise that modularity is purely about the physical design: it is the use of classes and objects that form the logical basis of the project.

Inheritance

Inheritance is in many ways an extension of the use of encapsulation and modularity, although it is concerned with abstraction. Generally, inheritance is a way of ranking or ordering abstractions into hierarchies. There are two main hierarchies in most complex systems, the Class structure (also known as ' Type-of ' or ' A Kind Of ') and the Object structure (also known as ' Part-of ' or ' Has-A '). Inheritance is concerned with the former, and its key feature is the inheritance of behaviour and state information.

More classes based on LightSource:

```
class LightSource
{
private:
    int lightState;

    virtual void Light(void) = 0;
    virtual void Dark (void) = 0;

public:
    LightSource() {lightState = 0;}
    ~LightSource();
    void TurnOn (void) { if(!lightState) Light(); lightState = 1;}
    void TurnOff (void) { if(lightState) Dark(); lightState = 0;}
    int IsOn(void) {return (lightState != 0);}
};

class LightBulb : public LightSource
{
    private:
        virtual void Light(void); // still need to be defined
        virtual void Dark (void); // .......
};

class FluorescentLight : public LightSource
{
    private:
        virtual void Light(void); // still need to be defined
        virtual void Dark (void); // ........
};
```

Note that we have now admitted that there is no such operation as Light or Dark when applied to the class LightSource: it is an ABSTRACT CLASS (because it has at least one virtual function ' pure ' - indicated by = 0). Such a class cannot be instantiated, and must be derived from (inherited from). The virtual keyword ensures that the right version of the function is called if access

is via a pointer to a base class. In this example we have made Light and Dark virtual at all levels, so that the mechanism will work all the way down the hierarchy, that is, LightBulb and FluorescentLight can in turn be used as base classes.

A second feature of inheritance is the ability to build on base classes, adding functionality (with both functions and data) as required without losing the features of the base class. Say we now wanted to create a variable light source: one with a dimmer switch. Rather than starting from scratch (no existing light source has this ability) we would avoid losing functional compatibility with other light sources by building on the existing class:

```
class VariableLight : public LightSource
{
private:
    virtual void Light (void);
    virtual void Dark  (void);

    float lightIntensity;  //0 dimmest, 1.0 brightest
public:
    VariableLight() {lightIntensity = 1.0;}

    virtual void SetIntensity { float level};
    float LightLevel(void) { return lightIntensity;}
};
```

This new derived type still has the functions TurnOn and TurnOff thus presenting the same interface to the client. However, it has added the operation SetIntensity, declared here as virtual to allow this to be a usable base class, and the data item ('state value') lightIntensity and an access function LightLevel for reading it.

Inheritance Continued

Definition: Inheritance is a process or technique by which one class, the derived class, inherits the members and methods of another class, the base class. Additional members and methods are usually added to the derived class to make it more specialized.

Inheritance further models real world concepts by providing a natural classification of objects and for allowing the commonality of objects to be taken advantage of.

Inheritance is a relationship between class where one parent is the ancestor (or parent/whatever) of another class. It provides programming by extension rather than reinvention. For instance in UT2K3 we don’t need to have a Player class for each game type, rather we have a base class Player which is then extended by the different player classes for the different game types.

You may also see inheritance referred to in is-a or is-a-kind-of terms, this is just another way of describing the concepts.
These inherited classes when further inherited form what is called a hierarchy.

A good example and one used in SAMS "Teach yourself C++ in 24hrs", a rather optimistically titled book, is that of the way taxonomists often divide up life.
Life is divided into kingdom, phyla, class, order, family, genus, and species.
You can imagine this like:

Animal → Mammal → Feline → Lion

Where the concept Lion would inherit (gets the features from) from Feline, Feline from Mammal and so on.
These concepts may contain information like:
Mammal – breathes air, moves
Feline – Extend Claws, be evil
The ideas of the Feline extending claws etc would be added to the idea in mammal. So a Feline would breathe air, move, extend claws, be evil.

These classes further down the line that inherit, are also said to be derived.
This is a way of expressing an is-a relationship.
You derive the new class Feline from the class Mammal, therefore a class that adds new functionality to an existing class derives from the original class.
The original class is generally called the base class.
Often a base class has more than one derived class, for instance we could add tigers:

Animal → Mammal → Feline → Lion
→ Tiger

In this case both lions and tigers are derived from Felines.

C++ Inheritance and Class Scope

Inheritance raises some interesting questions about scope.

Every class in C++ has its own individual scope even when it inherits members from other classes. Those members that are explicitly defined in the class definition of a derived class belong to the scope of that class; those members which are members in virtue of inheritance belong to the scope of the super-class from which they were inherited.

The scope rules for C++ classes have a number of consequences.

1) Those member functions which belong to the scope of a derived class will only have access to the data members within the scope of the derived and to those data members which are inherited from the public or protected section of the super-class.

2) Constructors for a derived type must make use of super-class constructors in order to initialise the data members inherited from the super-class.

3) Where protection is sought for the data members of a superclass but access to those members required by a subclass, those members should be declared in the protected section of the superclass.

Three Kinds of Inheritance
C++ provides three forms of inheritance: private inheritance (the default), protected inheritance and public inheritance. Public inheritance corresponds to the 'IsA' relationship, private and protected inheritance have different and less readily describable meanings.

Private Inheritance

Base Class
Derived Class
Private Members become Private Members
Protected Members become Private Members
Public Members become Private Members

Protected Inheritance

Base Class
Derived Class
Private Members become Private Members
Protected Members become Protected Members
Public Members become Protected Members

Public Inheritance

Base Class
Derived Class
Private Members become Private Members
Protected Members become Protected Members
Public Members become Public Members

Multiple Inheritance
Single inheritance implies that a class has only one direct ' super-class ', multiple inheritance allows the derivation of a class from more than one base class. A potential problem for multiple inheritance occurs when a chain of inheritance derives from a single class via two or more paths, so causing duplication of inherited data and functions. To avoid this problem; use the scope resolution operator :: or include the keyword virtual (again) in the inheritance list:

```
class A
{
      public:
      int memberA; //etc.
};
```

```
class A1 : virtual public A
{
      // A1 members ...
};

class A2 : virtual public A
{
      // A2 members ...
};

class B : public A1, public A2
{
      // has access to one and only one memberA
}
```

In C++ it is perfectly possible that a class inherits fields and methods from more than one class simply by separating the different base classes with commas in the declaration of the derived class. For example, if we had a specific class to print on screen (COutput) and we wanted that our classes CRectangle and CTriangle also inherit its members in addition to those of CPolygon we could write:

```
class CRectangle: public CPolygon, public COutput
{
      //.. member data and functions here

}

class CTriangle: public CPolygon, public COutput;
{
      //.. member data and functions here

}
```

Constructors and Destructors

You may have noticed that there were two constructors above, one in Player and one in DMplayer. The reason for this is that constructors and Destructors are not inherited.
Each subtype has its own constructor and destructor. The reason for this is that each object of a subtype consists of multiple parts, a base class part and a subclass part. The base class constructor forms the base class part. The subclass constructor forms the subclass part. Destructors clean up their respective parts.

The protected keyword

Now we introduce a new one protected. Protected is very similar to private, with one exception. When we define classes derived from a base class they don't have access, except through member functions to private member data. We are therefore unable to define new functions that have access to this data.

With the protected keyword we can. Anything that is protected is accessible to any of the base class derived classes.

```
#include <iostream>

using namespace std;

class X
{
public:
   void setProtMemb( int i ) { m_protMemb = i; }
   void Display() { cout << m_protMemb << endl; }
protected:
   int  m_protMemb;
   void Protfunc() { cout << "\nAccess allowed\n"; }
} x;


class Y : public X {
public:
   void useProtfunc() { Protfunc(); }
} y;


int main()
{
   // x.m_protMemb;          error, m_protMemb is protected
   x.setProtMemb( 0 );   // OK, uses public access function
   x.Display();
   y.setProtMemb( 5 );   // OK, uses public access function
   y.Display();
   // x.Protfunc();          error, Protfunc() is protected
   y.useProtfunc();      // OK, uses public access function
                         // in derived class
}
```