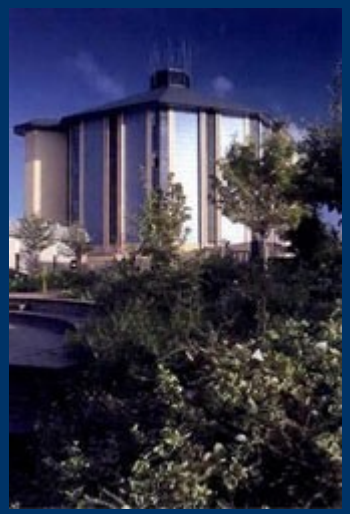
Inheritance & Classification Hierarchies

Building

Comercial

Public

Domestic



Inheritance & Classification Hierarchies

- Inheritance is one of the most powerful features of O-O programming
- By organising classes into Classification Hierarchies we can
    - add extra dimensions to encapsulation by grouping ADT's
    - and enables classes to inherit from other classes
    - thus extending the attributes and methods of the class which inherits
- The inheriting class may then also add extra functionality and attributes to produce a new object

Why do we need Inheritance ?

- What is the purpose of inheritance and why should we wish to inherit the attributes and methods of other Objects ?
- There are two main reasons for using inheritance in O-O which are
  - Specialisation
    - Extending the functionality of an existing class
  - Generalisation
    - Sharing commonality between two or more classes

What Do Objects Inherit ?

- A class does not contain any state values, only a 'blue print' for what value are to be contained
- Therefore a 'derived class' inherits all of the attributes from the 'base class'
- A derived class is by definition identical to the base class, but it can be built on to extend and modify the base class
- Objects of the derived class do not inherit anything from the objects of the base class
- As far as objects are concerned there is no hierarchy

Generalisation - sharing commonality

- When analysing a problem we sometimes find the following
  - Some Objects share the same attributes
  - Some Objects share the same methods
  - Some Objects require similar functionality

- So what is the best way of partitioning the problem ?
  - Lots of Classes some of which share the same attributes and methods
  - Fewer classes and avoid duplication but some Objects will have redundant data

Abstract Classes

- Sometimes the base class does not contain enough data to instantiate an object
- In this case the base class can be thought of as an 'abstract class'
- And the derived class will inherit the behaviour of the abstract base class and enough additional information to instantiate an object.
- However the base class doesn't necessarily have to be abstract

Inheritance Using C++

First we will define a simple Class called **`BaseClass`**

```
class BaseClass
{
private :
      int x;
public :
      void setX(int x_in){ x=x_in;}
      int getX(){ return x;}
};
```

- Base Class has one attribute an integer x
- Two methods to set and get the value of x
- This class can be instantiated in the usual way and the methods called on the class
- At present there is no inheritance as we only have one class

Adding a derived class

- We can extend the `BaseClass` by adding a new attribute y and methods `setY` and `getY`
- The derived class will now have two attribute X and Y and associated methods
- The syntax for a derived class to inherit the base class is as follows

```
class derived_class_name:
      public/private base_class_name
```

Adding a derived class

- The derivation type may be either public or private
- Public is the most common as it allows
  - Objects of the derived class access to the public parts (usually methods)
  - as well as the public parts of it own class
- Private is less common, and means that the derived class object may only use the methods defined in the derived class, not those inherited from the base class. However derived class methods may utilise base class methods to implement their own behavior

DerivedClass Example

- The following DerivedClass definition demonstrates the syntax for inheritance

```
class DerivedClass : public BaseClass
{
private :
    int y;
public :
    void setY(int y_in){y=y_in;}
    int getY() { return y;}
};
```

DerivedClass Example

- The following code demonstrates the use of the two classes

```
#include <iostream.h>
void main(void)
{
    BaseClass base_object;
    DerivedClass derived_object;
    base_object.setX(7);
    derived_object.setX(12);
    derived_object.setY(1);
}
```

The protected inheritance

- To allow a derived class to access the attributes of the base class the "**private**" keyword is replaced with "**protected**" keyword

```
class BaseClass
{
protected :
      int x;
.......



class DerivedClass : public BaseClass
{
.........
void xEqualsy(){ y=x;  }
};
```

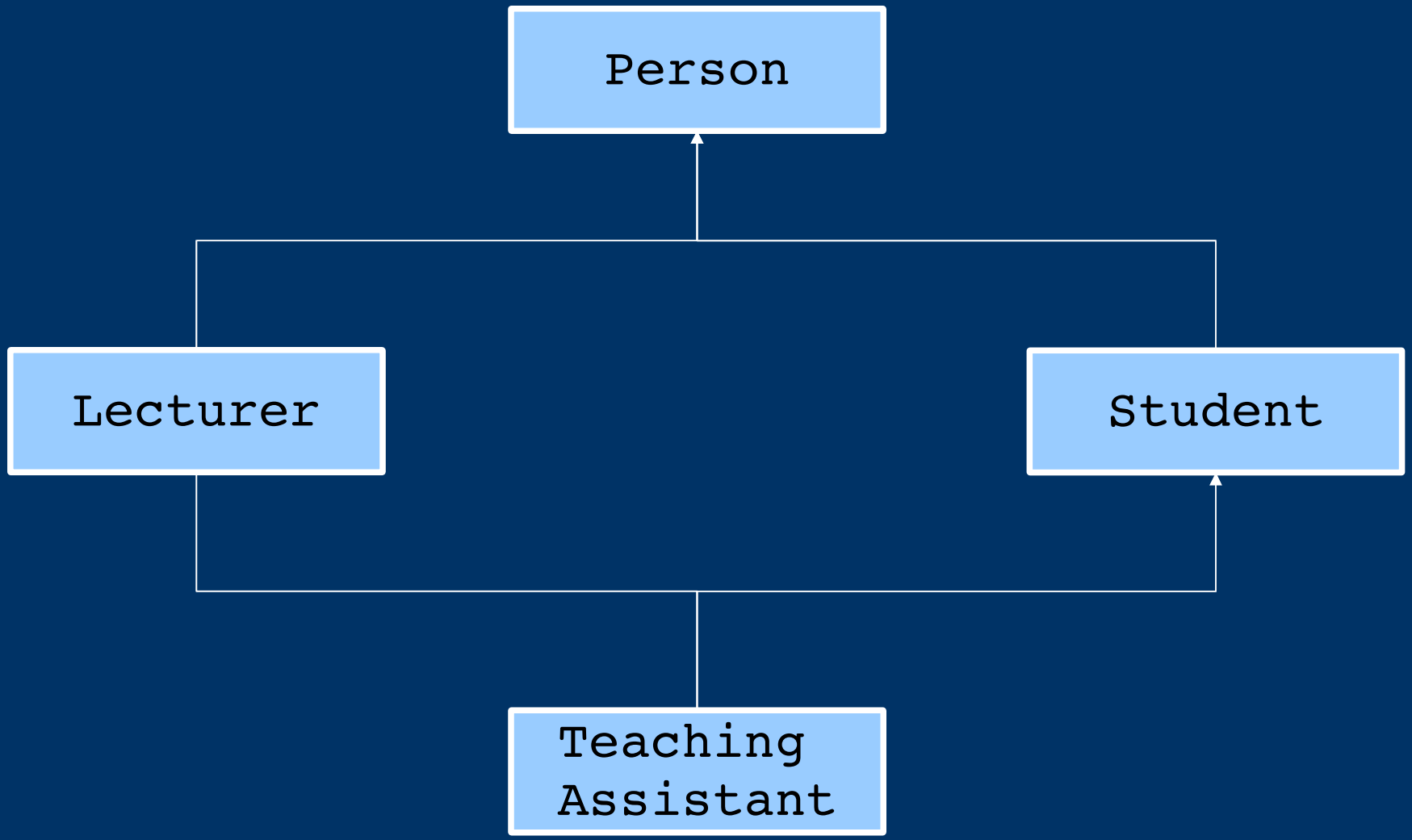
Inheriting Constructors

- A derived class will always inherit the constructor of the base class
- A derived class will also have it's own constructor
- The base class constructor is called first followed by each derived class going down the hierarchy tree
- This is because each class constructor must allocate the memory required for the attributes of the class
- If the constructor for the base class has parameters then the derived class must have a constructor of the same type so that it may follow up the hierarchy tree

Inheriting destructors

- Derived Classes also inherit the base classes destructors
- However destructors are called in reverse order to the constructors
- So all the derived classes destructors are called first
- And the base class destructor is called last
- As destructors have no parameters there is now problem with the definition of correct parameter lists

An Example of Multiple Inheretance



Multiple Inheritance

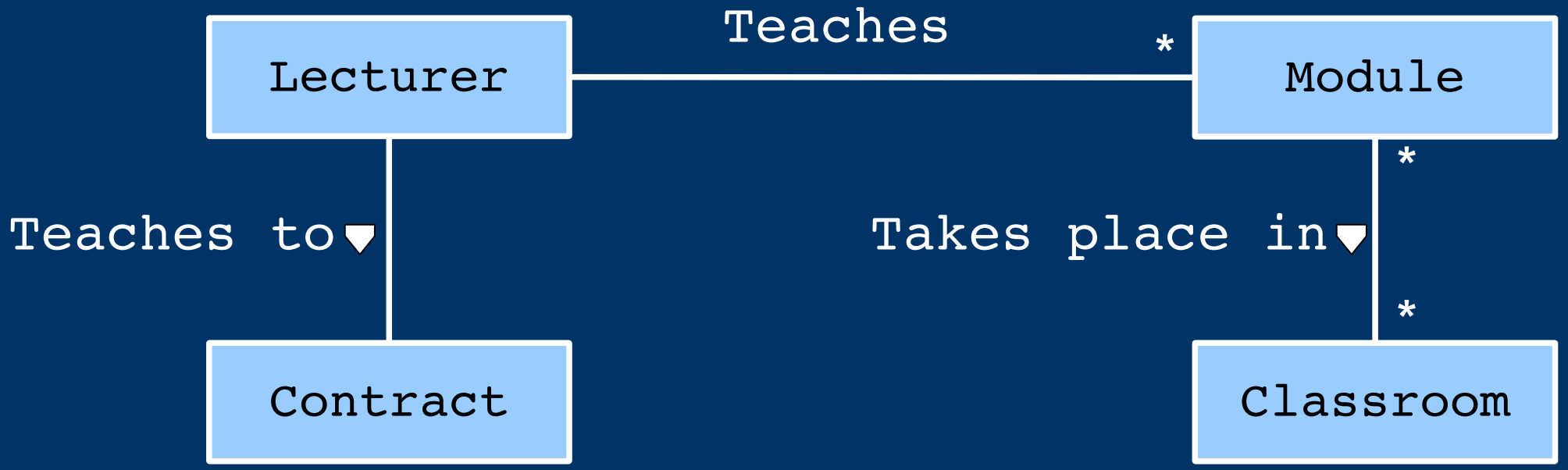
- It is possible for a class to have more than one base class

```
class DerivedClass : public    BaseClass-1,
                     public    BaseClass-2,
                     private   BaseClass-3,
                     protected BaseClass-4,
{

};
```

- The base classes must be distinct..
- The order is insignificant

Association & Agregation



The Association in a timetable example

Aggregation & Composition

Aggregation is a relationship in which one object is a part of another.

A aggregates B

=

B is part of A, but their lifetimes may be different

Ex: cars and wheels, engine, etc.

Composition is a relationship in which one object is an integral part of another

A composes B

=

B is part of A, and their lifetimes are the same

Ex: person and brain, lung, etc.

Aggregation v. Inheritance

- In this type of hierarchy, classes do not inherit from other classes but are composed of other classes
- This means that an object of one class may have it's representation defined by other objects rather than by attributes.
- The enclosing class does not inherit any attributes or methods from these other included classes.
- An object of the enclosing class is composed wholly or partly of objects of other classes
- **Any object that has this characteristic is know as an aggregation**

A Containment Example

- For Example A Car
  - will have an engine compartment with an integral component :- the engine
  - This is a containment relationship as the engine is an essential part of a car
- Another part of a car is the boot. What is in the boot does not effect the integrity of the car (i.e. what the car is)
  - The boot can contain many different types of objects (tools, shopping etc) but this does not affect the car object.
- Therefore the boot is a container existing independently of it's contents.

Aggregation C++ Syntax

- There are two basic ways in which associations and aggregations are implemented

1. Objects contain Objects
2. Objects contain pointers to Objects

- The first approach is used to create fixed aggregations (objects inside objects)
- The second is used to create variable aggregations to make programs more flexible

Aircraft Class: Fixed Aggregation

```
class Aircraft
{
private :
    PortWing port_wing;
    StarboardWing starboard_wing;
    Engine engine1,engine2;
    Fuselage fuselage;
    Tailplane tailplane;
};
```

- Each of the separate classes within the Aircraft class will have their own methods which can be called within the Aircraft class as shown below

Example of Methods Aircraft Class

```
void Aircraft::turnToPort()
{
 port_wing.elevatorUp();
 starboard_wing.elevatorUp();
 port_wing.aileronUp();
 starboard_wing.aileronDown();
 tailplane.rudderLeft();
}
```

- Activities of some composing objects will depend on the state of others

Example of Methods Aircraft Class

```
void Aircraft::openDoors()
{
if(engine1.getSpeed()>IDLE||engine2.getSpeed()>IDLE)
   {
     //don't open doors
   }
else
    fuselage.openDoors();
}
```

- This is an example of a propagation as the state of the Aircraft propagates to the state of the door.

Car Class Comprised of wheel and engine

```
class Wheel
{
private:
    int diameter;
public:
    Wheel(int diameter_in){ diameter = diameter_in; }
    int getDiameter() { return diameter; }
};
class Engine
{
private:
        int cc;
public:
        Engine(int cc_in)  { cc = cc_in; }
        int getCC()return cc; }
};
```

Car Class

```
class Car
{
private:
        Wheel nearside_front, offside_front,
        nearside_rear, offside_rear;
        Engine engine;
        int passengers;
public:
  Car(int diameter_in, int cc_in, int passengers_in);
  void showSelf();

};
```

Car Class

```
Car::Car(int diameter_in, int cc_in, int
passengers_in) :
          nearside_front(diameter_in),
          offside_front(diameter_in),
          nearside_rear(diameter_in),
          offside_rear(diameter_in),
          engine(cc_in)
{
        passengers = passengers_in;
}
```