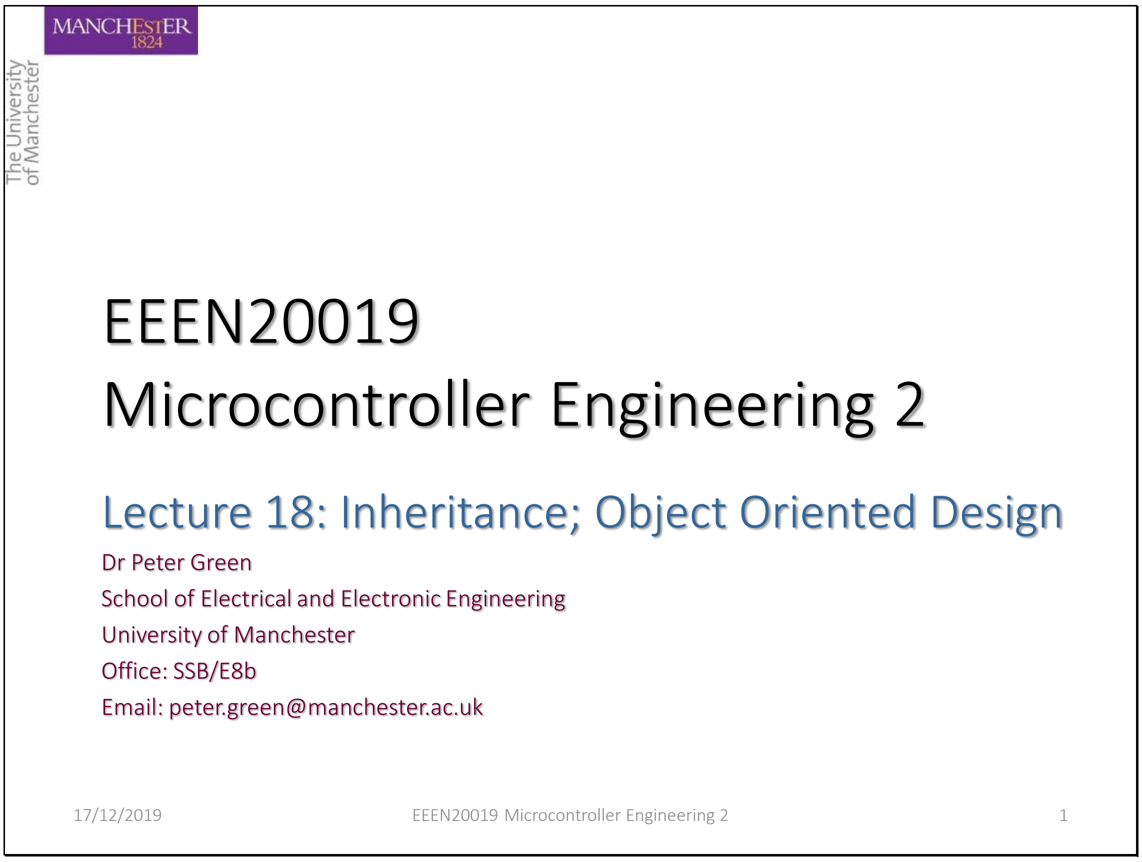
EEEN 20019
Microcontroller Engineering 2



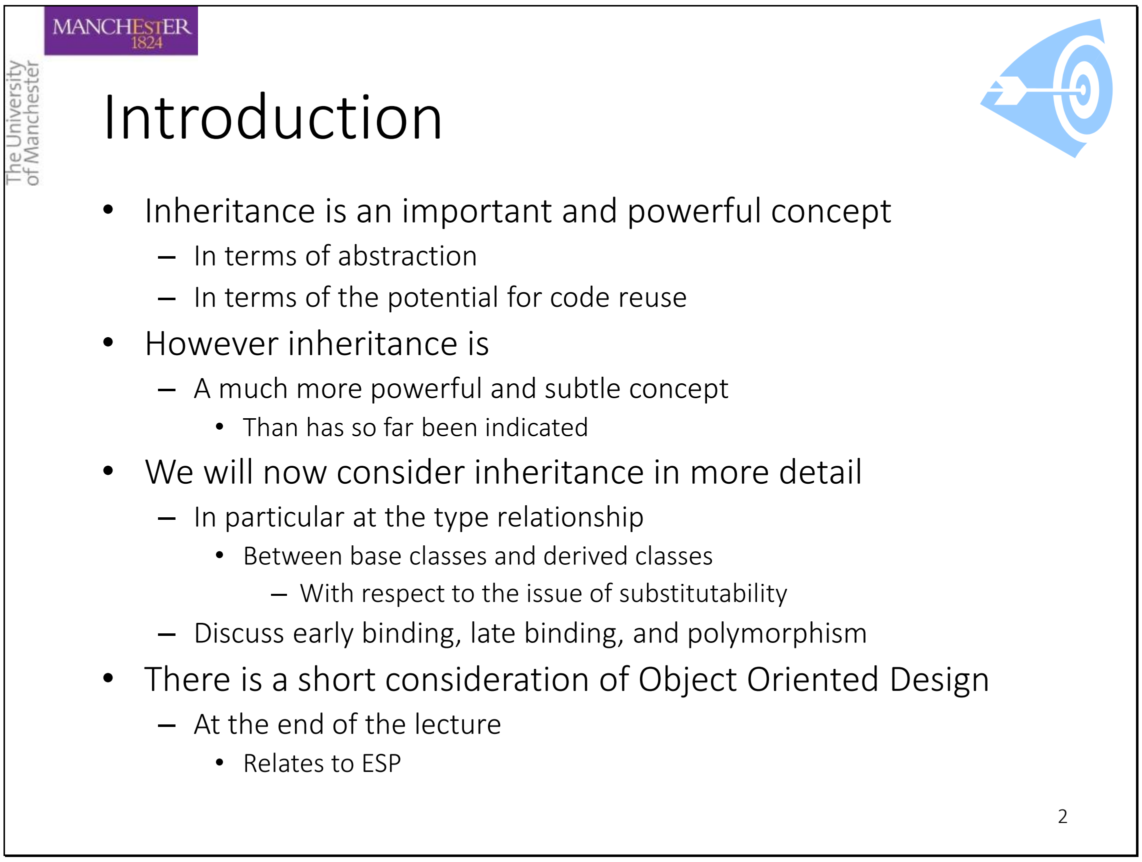
EEEN20019
Microcontroller Engineering 2

Lecture 18: Inheritance; Object Oriented Design

Dr Peter Green
School of Electrical and Electronic Engineering
University of Manchester
Office: SSB/E8b
Email: peter.green@manchester.ac.uk

17/12/2019 EEEN20019 Microcontroller Engineering 2 1

Dr. Peter Green
School and Electrical and Electronic Engineering
University of Manchester
peter.n.green@manchester.ac.uk



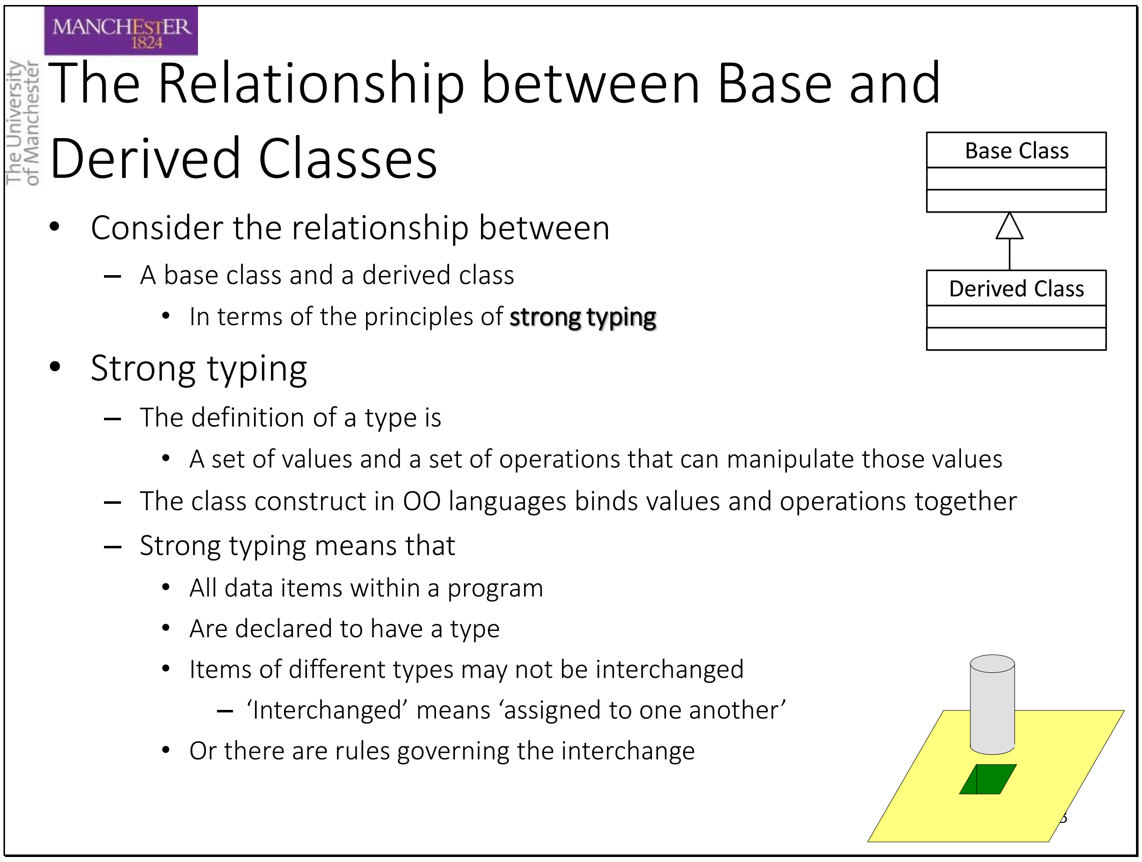
MANCHESTER 1824

The University of Manchester

Introduction

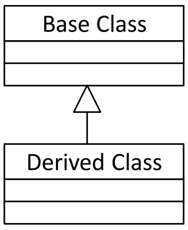
- Inheritance is an important and powerful concept
  - In terms of abstraction
  - In terms of the potential for code reuse
- However inheritance is
  - A much more powerful and subtle concept
    - Than has so far been indicated
- We will now consider inheritance in more detail
  - In particular at the type relationship
    - Between base classes and derived classes
      - With respect to the issue of substitutability
  - Discuss early binding, late binding, and polymorphism
- There is a short consideration of Object Oriented Design
  - At the end of the lecture
    - Relates to ESP

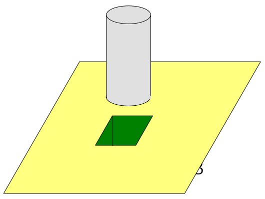
2



The Relationship between Base and Derived Classes

- Consider the relationship between
  - A base class and a derived class
    - In terms of the principles of **strong typing**
- Strong typing
  - The definition of a type is
    - A set of values and a set of operations that can manipulate those values
  - The class construct in OO languages binds values and operations together
  - Strong typing means that
    - All data items within a program
    - Are declared to have a type
    - Items of different types may not be interchanged
      - 'Interchanged' means 'assigned to one another'
    - Or there are rules governing the interchange



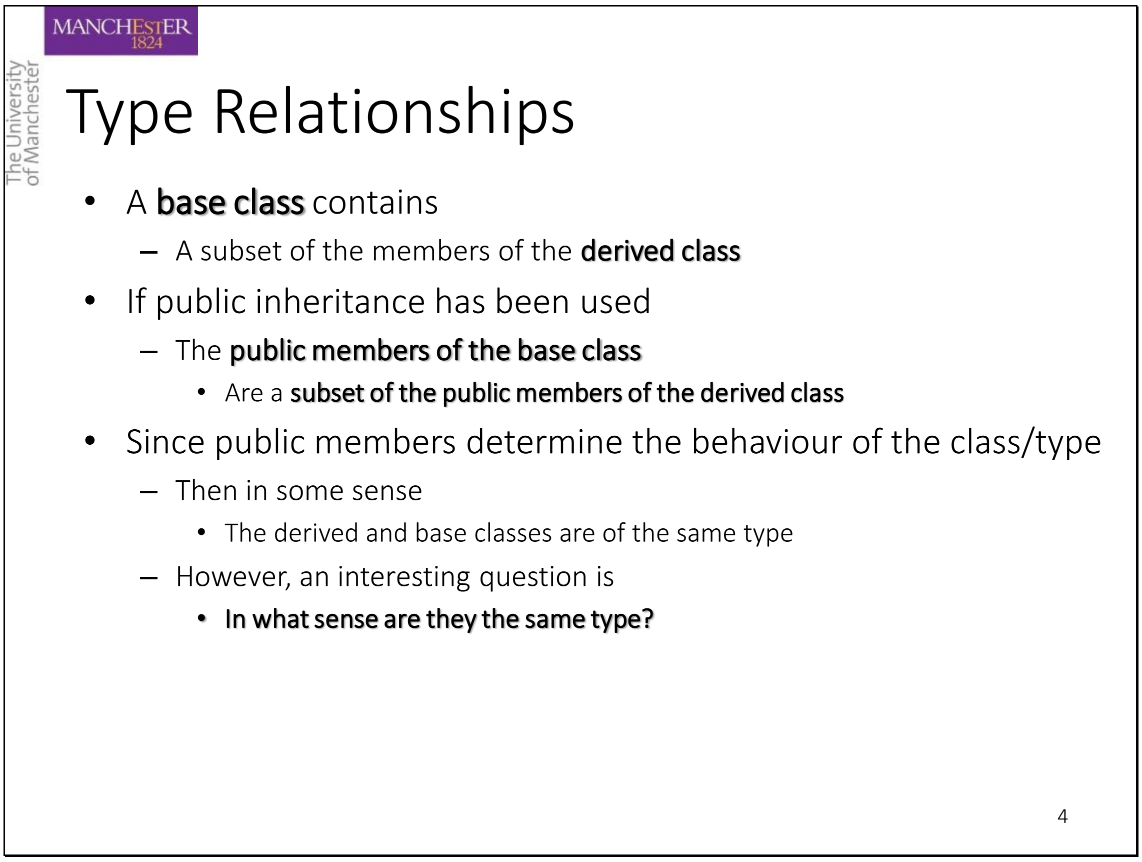


The exact relationship between the concepts of type and class is rather esoteric. For our purposes, we will view class as an extension of the concept of type familiar from the first year. Hence, even in C, which is weakly typed, all variables in a program must be declared to be of some type, and there are rules about how variables of specific types can be used e.g. what they can and cannot be assigned to, what operators may be applied to them etc etc. Languages are said to be strongly typed or weakly typed depending upon how rigorously these rules are upheld by the compiler. C is weakly typed because the type rules may be circumvented fairly easily. Strong typing typically provides more security from error, as the compiler is more able to detect the misuse of variables. This is typically at the expense of flexibility. However, there are circumstances under which strong typing is unhelpful, and even strongly types languages usually provide some mechanism for overriding the typing rules.

Strong typing can be viewed as the enforcement of the class of an object, such that objects of different types may not be interchanged, or at most, they may be interchanged in very restricted ways.

Typing enables abstractions to be created, and for the implementation programming language to enforce these design decisions.

See G. Booch : Object-Oriented Analysis and Design with Applications, page 66, 2nd edition, Benjamin-Cummings1994.

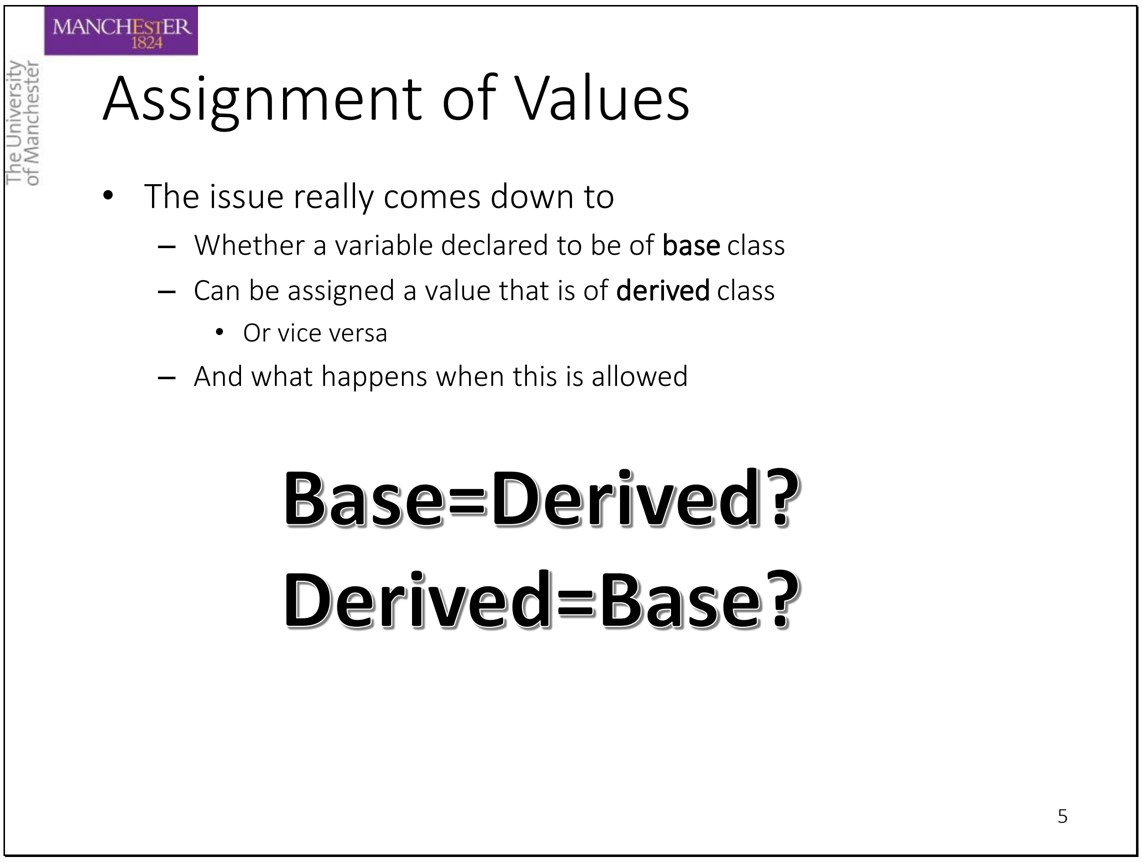


Type Relationships

- A **base class** contains
  - A subset of the members of the **derived class**
- If public inheritance has been used
  - The **public members of the base class**
    - Are a **subset of the public members of the derived class**
- Since public members determine the behaviour of the class/type
  - Then in some sense
    - The derived and base classes are of the same type
  - However, an interesting question is
    - **In what sense are they the same type?**

4

To simplify the above, if classes have common aspects, are they the same type? This is interesting considering the definition of a type as the set of values that a variable of the type can take. If this is applied, then clearly the set of values of the base class is a subset of the set of values of the derived class. This relationship is also true of the relationship between operations (MFs). The operations that apply to the base class are a subset of those applicable to the derived class.



Assignment of Values

- The issue really comes down to
  - Whether a variable declared to be of **base** class
  - Can be assigned a value that is of **derived** class
    - Or vice versa
  - And what happens when this is allowed

Base=Derived?

Derived=Base?

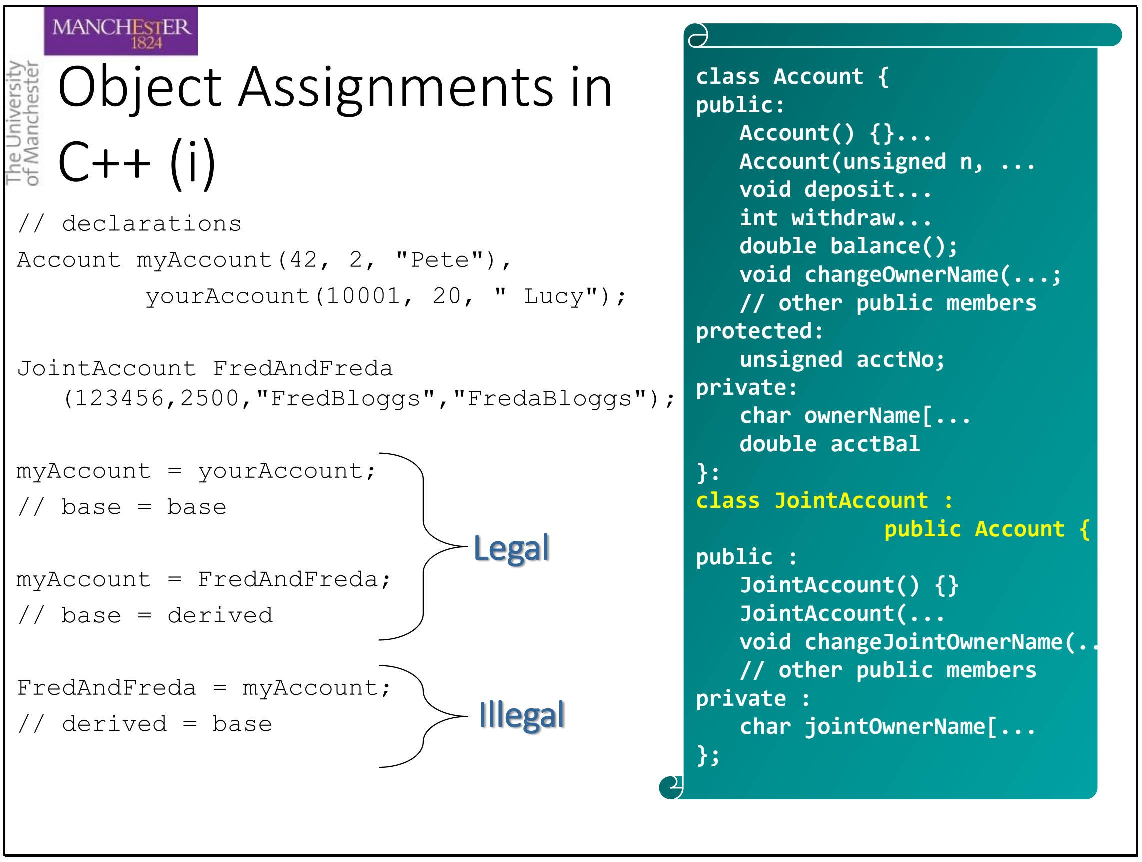
5

This question is based on the previous slide. As considered above, an interesting way of thinking about this is in terms of sets.

If the set of base class values is a subset of the set of derived class values, then an assignment function between the two classes works for **the set intersection** of derived set values with those of the base class. However, what about assignments between base class class and those derived class variables that are in the set difference of base and derived classes?

As an abstract example, consider that the B is the base class, and that B's values are B = {2, 4, 6, 8}. The derived class values are D = {2, 4, 6, 8, 10, 12, 14, 16}. Under what circumstances can a variable of type B be assigned to a variable of type D, and vice versa?

Before these questions are answered, a C++ code-level example is considered next.



Object Assignments in C++ (i)

```
// declarations
Account myAccount(42, 2, "Pete"),
        yourAccount(10001, 20, " Lucy");

JointAccount FredAndFreda
   (123456,2500,"FredBloggs","FredaBloggs");

myAccount = yourAccount;
// base = base

myAccount = FredAndFreda;
// base = derived

FredAndFreda = myAccount;
// derived = base
```

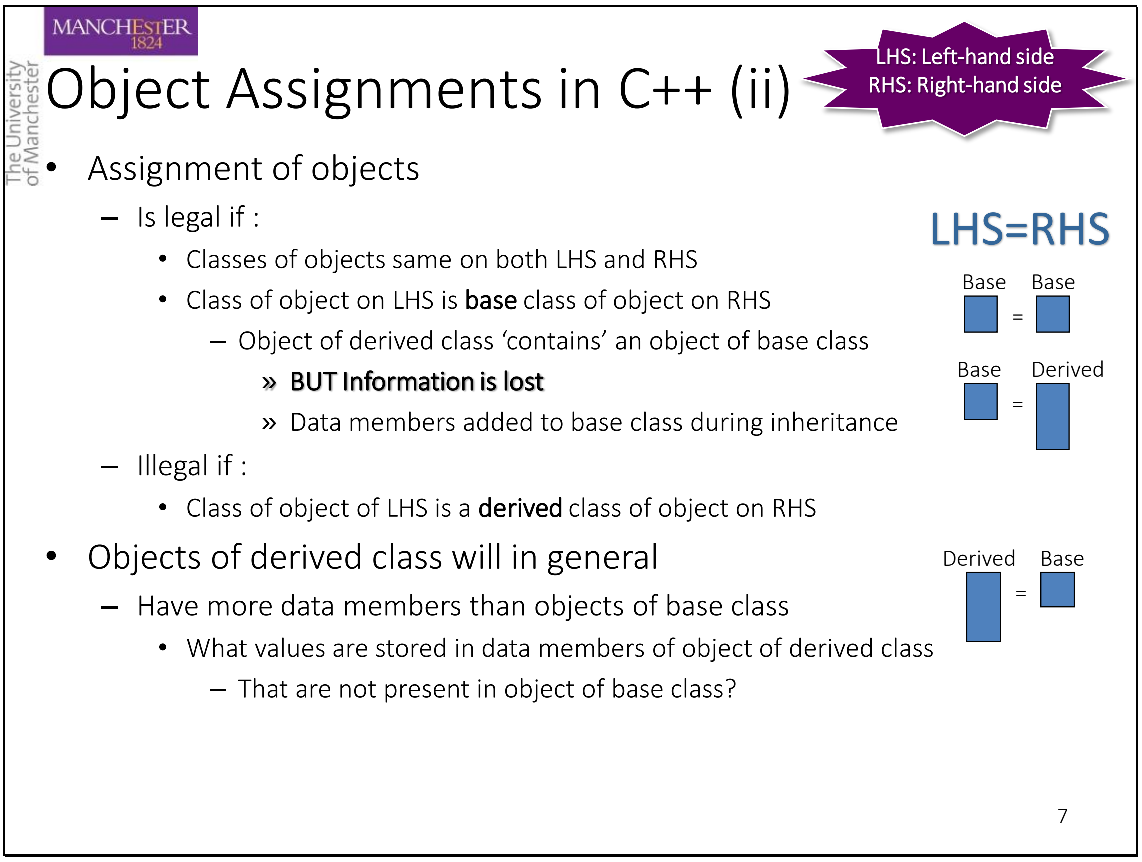
Legal
Illegal

```
class Account {
public:
   Account() {}...
   Account(unsigned n, ...
   void deposit...
   int withdraw...
   double balance();
   void changeOwnerName(...;
   // other public members
protected:
   unsigned acctNo;
private:
   char ownerName[...
   double acctBal
}:
class JointAccount :
             public Account {
public :
   JointAccount() {}
   JointAccount(...
   void changeJointOwnerName(..
   // other public members
private :
   char jointOwnerName[...
};
```

Here the issue is assignments between objects of class `Account` and those of derived class `JointAccount`.

Consider the assignment to a base class (B) object (object is on the left hand side of an assignment). An assignment is allowed if:

- The right-hand side evaluates to an object of the same class
- The right-hand side evaluates to an object that is of the same class, or of a derived class

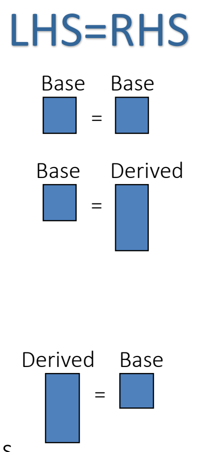


The University of Manchester

Object Assignments in C++ (ii)

LHS: Left-hand side
RHS: Right-hand side

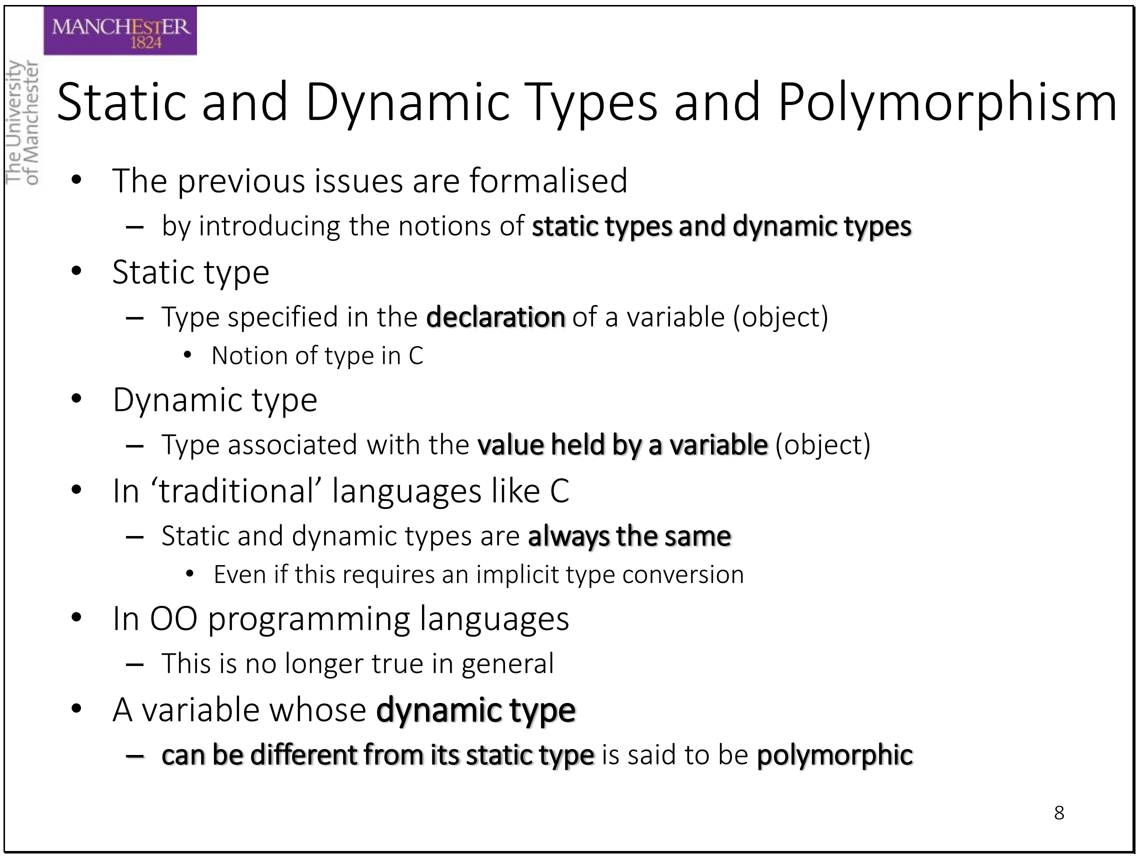
- Assignment of objects
  - Is legal if :
    - Classes of objects same on both LHS and RHS
    - Class of object on LHS is **base** class of object on RHS
      - Object of derived class 'contains' an object of base class
        - » **BUT Information is lost**
        - » Data members added to base class during inheritance
  - Illegal if :
    - Class of object of LHS is a **derived** class of object on RHS
- Objects of derived class will in general
  - Have more data members than objects of base class
    - What values are stored in data members of object of derived class
      - That are not present in object of base class?



7

The above slides illustrate the assignment rules for objects with a base class/derived class relationship. As indicated, an object of derived class can be assigned to one of base class, **but information is typically lost**. This is termed **slicing**, since during assignment the data members added to construct the derived class are sliced off and lost.

The straightforward way to remember these rules is via 'size'. Objects of derived class are 'bigger' than those of base class (they typically add data members to the base class). Hence, we can slice off the additional members to get a fit. If we try to do the assignment the other way round (i.e. assign an object of base class to one of derived class) then, since the object on the right-hand side is 'bigger' than the object on the left-hand side, then there is some extra 'room' inside the object after assignment. A more rigorous way of putting this is to say that some of the data members on the left-hand side will not be assigned to during the overall object assignment. Hence the question arises: what values should they hold. As there is no consistent answer to this question, then this form of assignment is forbidden.



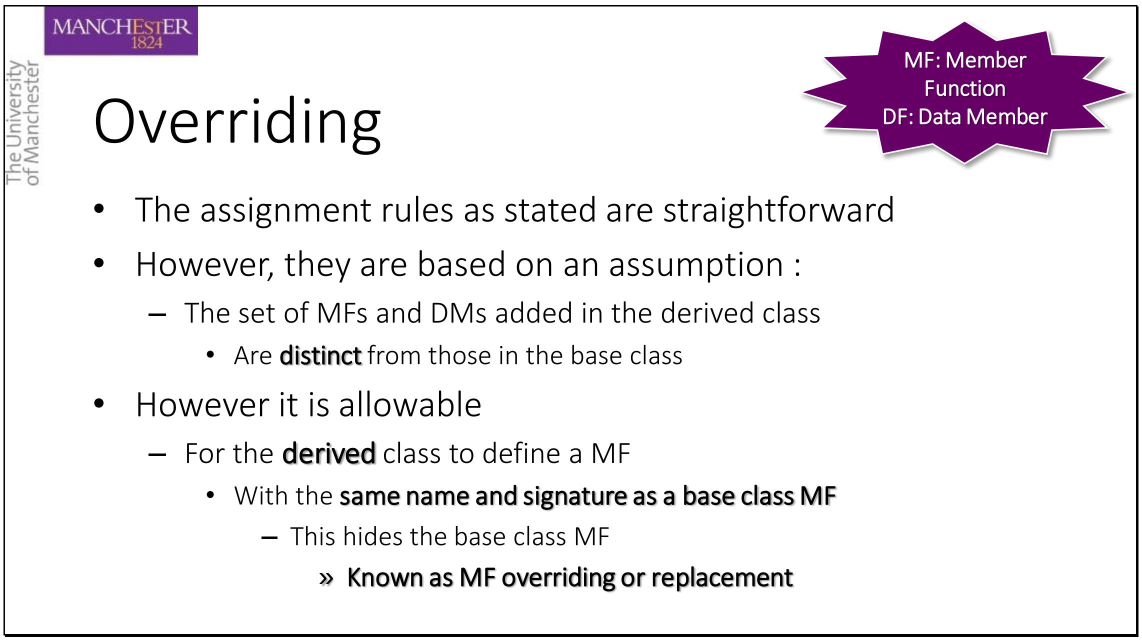
Static and Dynamic Types and Polymorphism

- The previous issues are formalised
  - by introducing the notions of **static types and dynamic types**
- Static type
  - Type specified in the **declaration** of a variable (object)
    - Notion of type in C
- Dynamic type
  - Type associated with the **value held by a variable** (object)
- In 'traditional' languages like C
  - Static and dynamic types are **always the same**
    - Even if this requires an implicit type conversion
- In OO programming languages
  - This is no longer true in general
- A variable whose **dynamic type**
  - **can be different from its static type** is said to be **polymorphic**

8

To be clear: the **static type** is the type specified in a variable's declaration, which is associated with the storage allocated to the variable. The **dynamic type** refers to the type of the value that is stored in the memory location(s) allocated to the variable.

The key issue here is the allowed relationship between static and dynamic types. C++ only allows the static and dynamic types of an object to differ **if the dynamic type is a subclass of the static type of the object**. This simply formalises the examples and rules that were given in the slides entitled 'Object Assignment in C++ (i) and (ii)'.

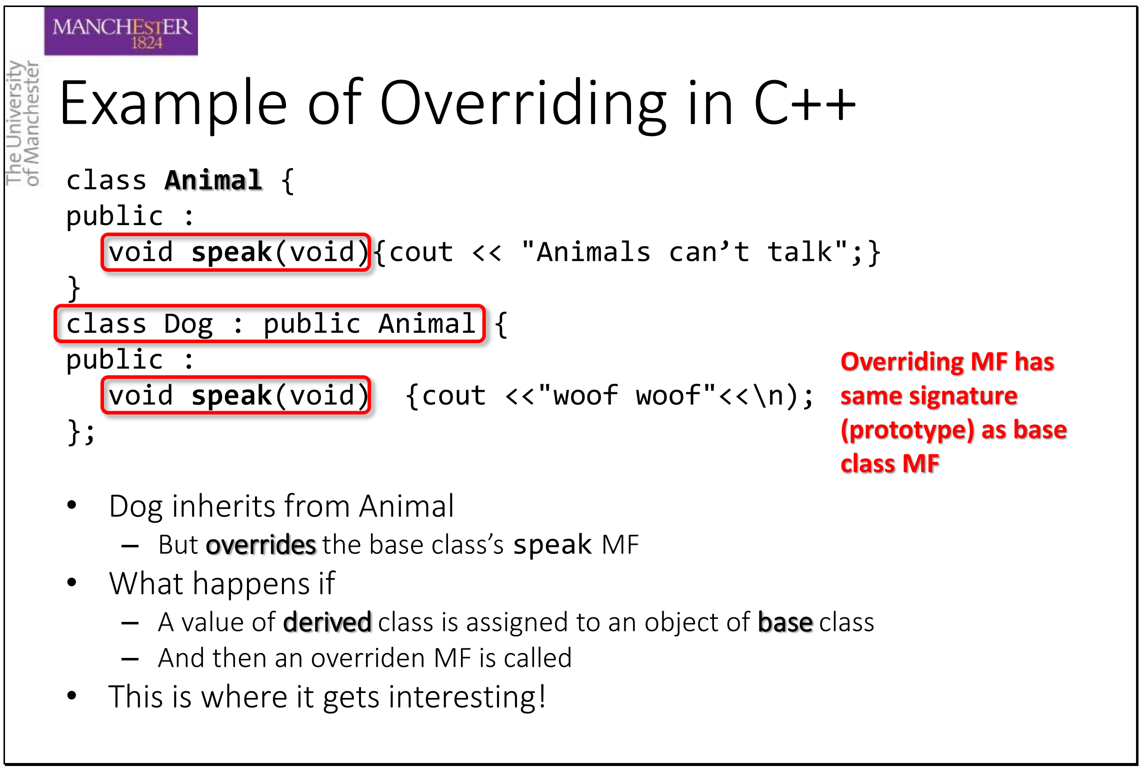


Overriding

MF: Member Function
DF: Data Member

- The assignment rules as stated are straightforward
- However, they are based on an assumption :
  - The set of MFs and DMs added in the derived class
    - Are **distinct** from those in the base class
- However it is allowable
  - For the **derived** class to define a MF
    - With the **same name and signature as a base class MF**
      - This hides the base class MF
        - » **Known as MF overriding or replacement**

There was an example of overriding in the first lecture on inheritance, where the withdraw member function from base class `Account` was overridden in the derived class `ChequeAcount`, in order to provide functionality associated with fee charging.

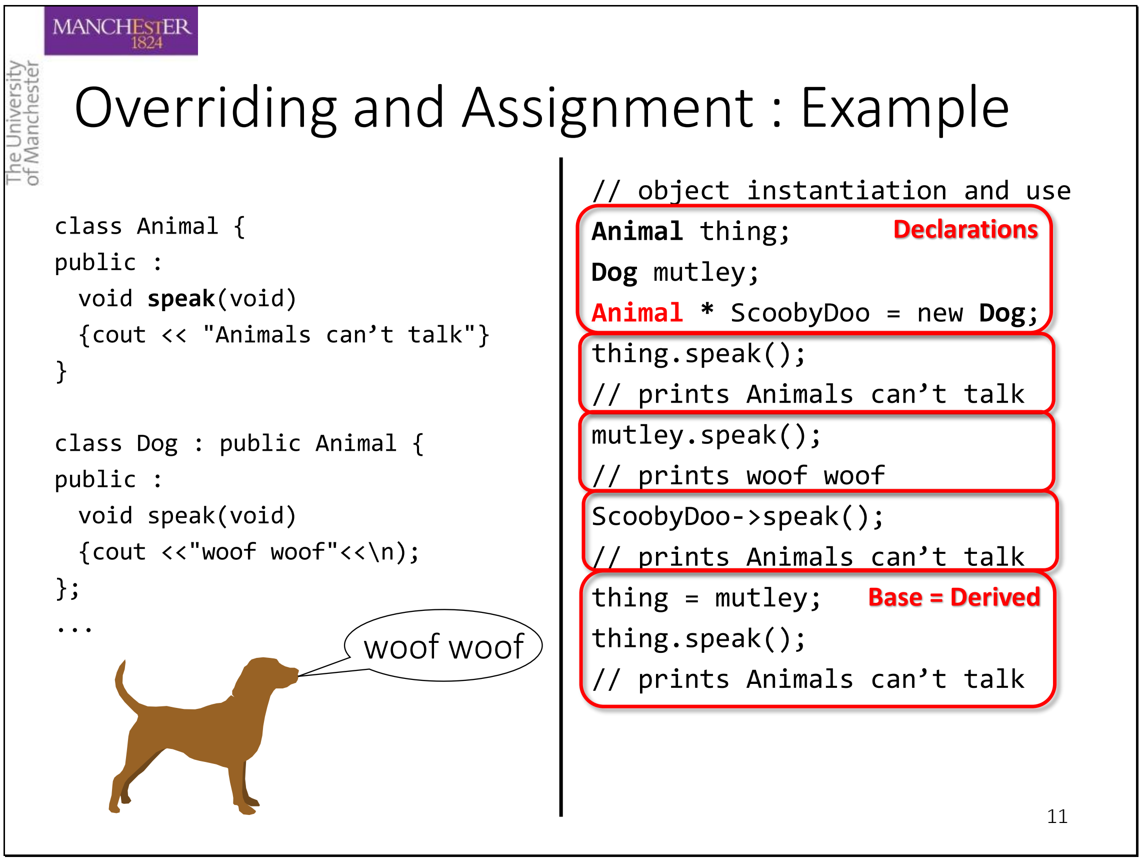


Example of Overriding in C++

```
class Animal {
public :
   void speak(void){cout << "Animals can’t talk";}
}
class Dog : public Animal {
public :
   void speak(void)  {cout <<"woof woof"<<\n);
};
```

Overriding MF has same signature (prototype) as base class MF

- Dog inherits from Animal
  - But **overrides** the base class’s `speak` MF
- What happens if
  - A value of **derived** class is assigned to an object of **base** class
  - And then an overriden MF is called
- This is where it gets interesting!

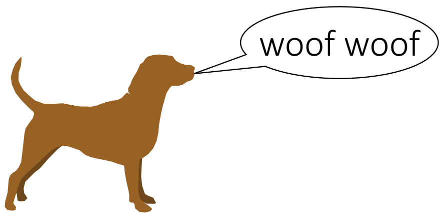


The University of Manchester

MANCHESTER 1824

Overriding and Assignment : Example

```
class Animal {
public :
  void speak(void)
  {cout << "Animals can’t talk"}
}

class Dog : public Animal {
public :
  void speak(void)
  {cout <<"woof woof"<<\n);
};
...
```



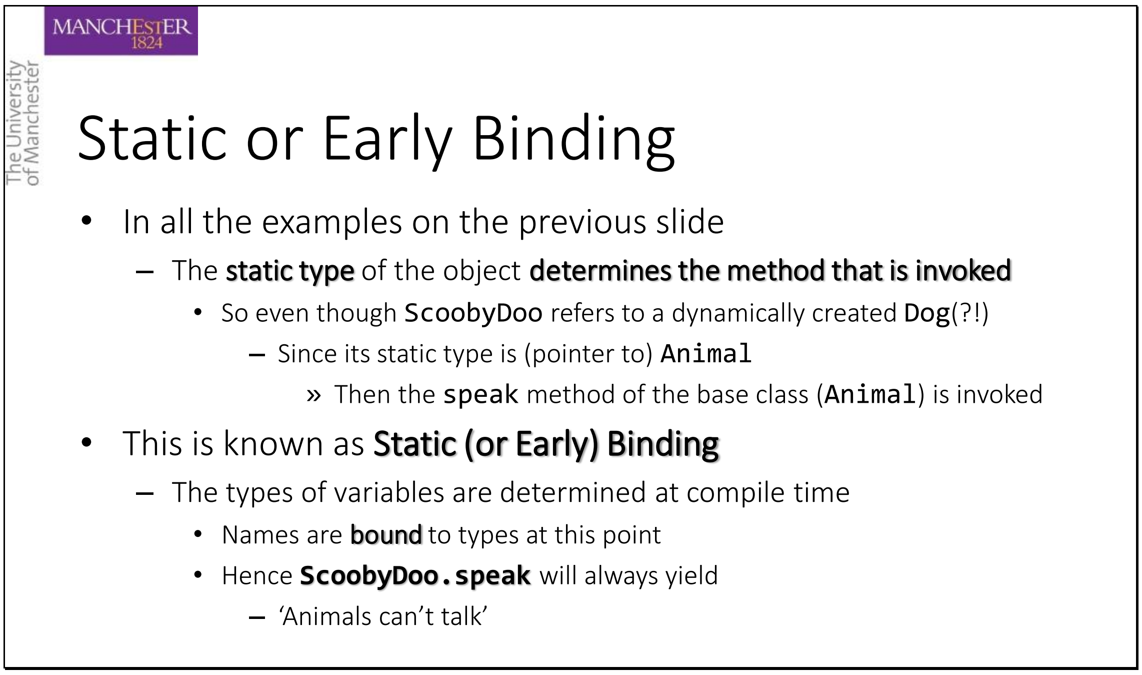
```
// object instantiation and use
Animal thing;          Declarations
Dog mutley;
Animal * ScoobyDoo = new Dog;
thing.speak();
// prints Animals can’t talk
mutley.speak();
// prints woof woof
ScoobyDoo->speak();
// prints Animals can’t talk
thing = mutley;      Base = Derived
thing.speak();
// prints Animals can’t talk
```

11

Here the expected behaviour for objects `thing` and `mutley` is clear. The class Dog has overriden the `speak` method from the base class `Animal`. However, as `mutley` is of class Dog, then the appropriate version of `speak` is invoked.

The behaviour of ScoobyDoo is also consistent with what has been said already. ScoobyDoo is a pointer to an object of base class (`Animal`), which is made to point to an object of derived class (Dog). This represents the assignment of an object of derived class to an object of base class via a level of indirection.

However, the interesting thing about this example is that the method invoked by `ScoobyDoo->speak` is determined by the static type of `ScoobyDoo`, and not by its dynamic type.


The University of Manchester

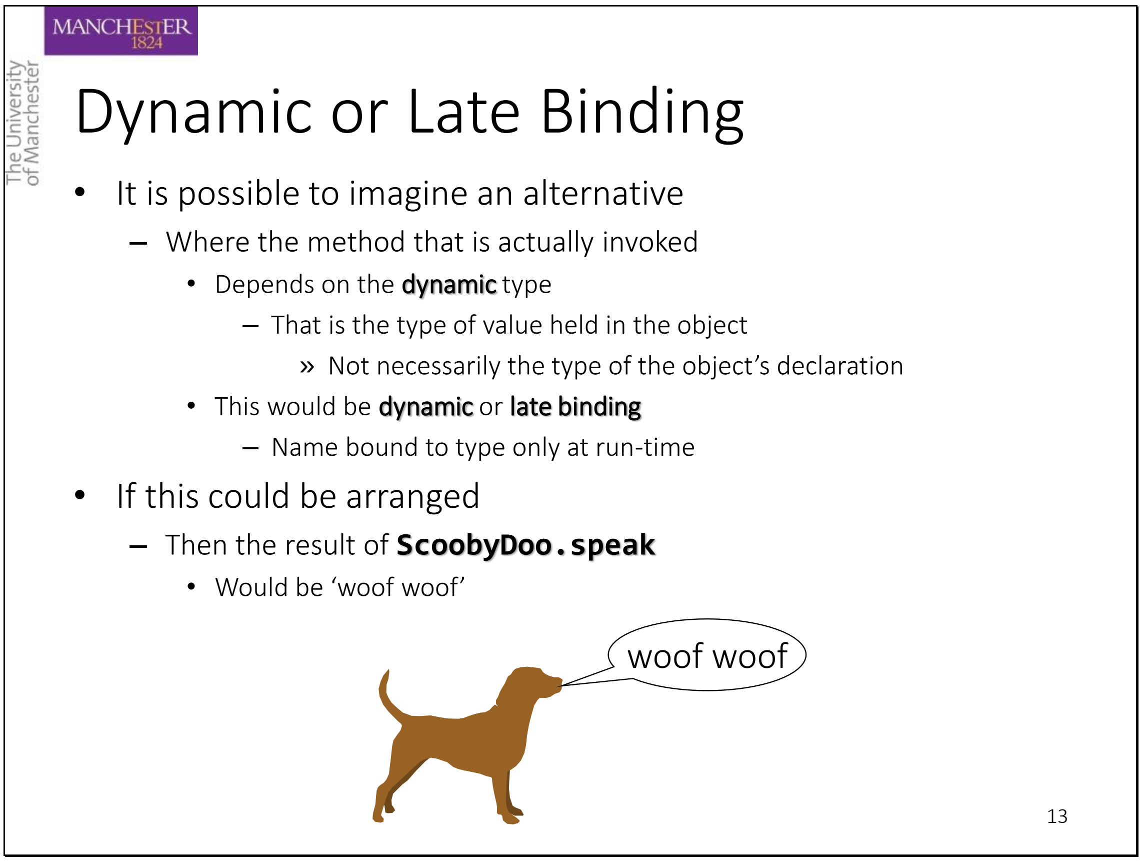
MANCHESTER 1824

Static or Early Binding

- In all the examples on the previous slide
  - The **static type** of the object **determines the method that is invoked**
    - So even though `ScoobyDoo` refers to a dynamically created `Dog`(?!)
      - Since its static type is (pointer to) `Animal`
        - » Then the `speak` method of the base class (`Animal`) is invoked
- This is known as **Static (or Early) Binding**
  - The types of variables are determined at compile time
    - Names are **bound** to types at this point
    - Hence **`ScoobyDoo.speak`** will always yield
      - 'Animals can't talk'

The term '**binding'** refers to the point at which the variable name is associated (or bound) with the specific object (**memory location**). If this is done at compile-time then we have **early** or **static binding**.

https://medium.com/omarelgabrys-blog/the-story-of-object-oriented-programming-12d1901a1825



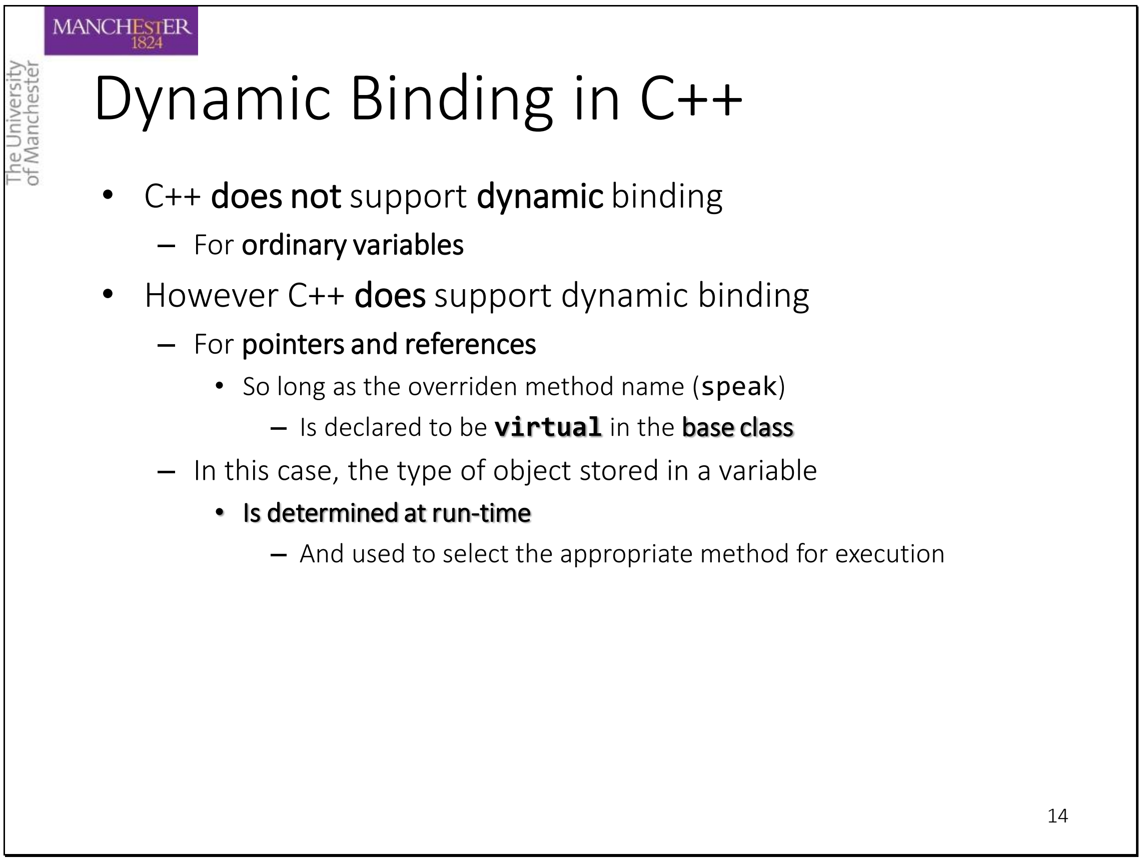
Dynamic or Late Binding

- It is possible to imagine an alternative
  - Where the method that is actually invoked
    - Depends on the **dynamic** type
      - That is the type of value held in the object
        - » Not necessarily the type of the object's declaration
    - This would be **dynamic** or **late binding**
      - Name bound to type only at run-time
- If this could be arranged
  - Then the result of **`ScoobyDoo.speak`**
    - Would be 'woof woof'

13

In late binding, an object of **derived** type can be stored in a **base** class variable (e.g. `thing = mutley` above). From earlier, it should be understood that, in some sense, a derived class is of the same type as the base class, so this makes sense. The really interesting question is if this is done, and an overridden member function is called, what happens?

Read on...



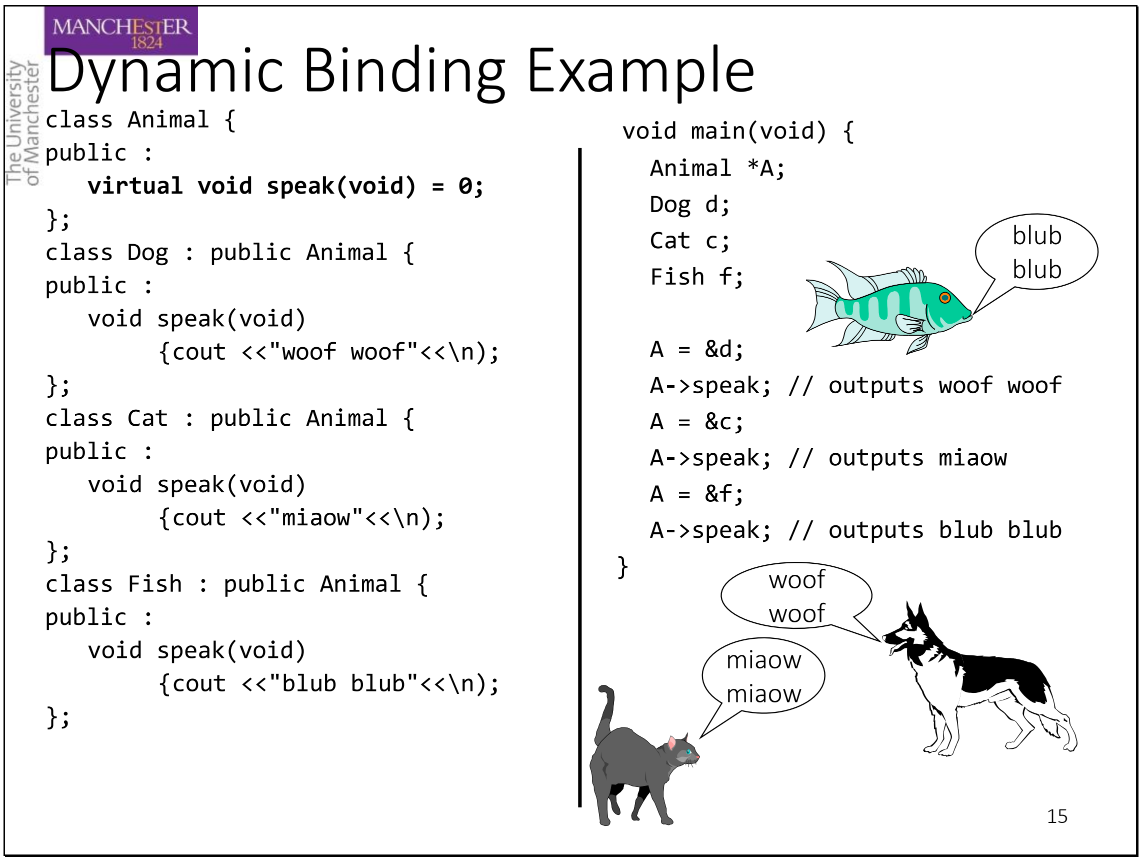
MANCHESTER 1824

The University of Manchester

Dynamic Binding in C++

- C++ **does not** support **dynamic** binding
  - For **ordinary variables**
- However C++ **does** support dynamic binding
  - For **pointers and references**
    - So long as the overriden method name (`speak`)
      - Is declared to be **`virtual`** in the **base class**
  - In this case, the type of object stored in a variable
    - **Is determined at run-time**
      - And used to select the appropriate method for execution

14



Dynamic Binding Example

```
class Animal {
public :
   virtual void speak(void) = 0;
};
class Dog : public Animal {
public :
   void speak(void)
        {cout <<"woof woof"<<\n);
};
class Cat : public Animal {
public :
   void speak(void)
        {cout <<"miaow"<<\n);
};
class Fish : public Animal {
public :
   void speak(void)
        {cout <<"blub blub"<<\n);
};
```

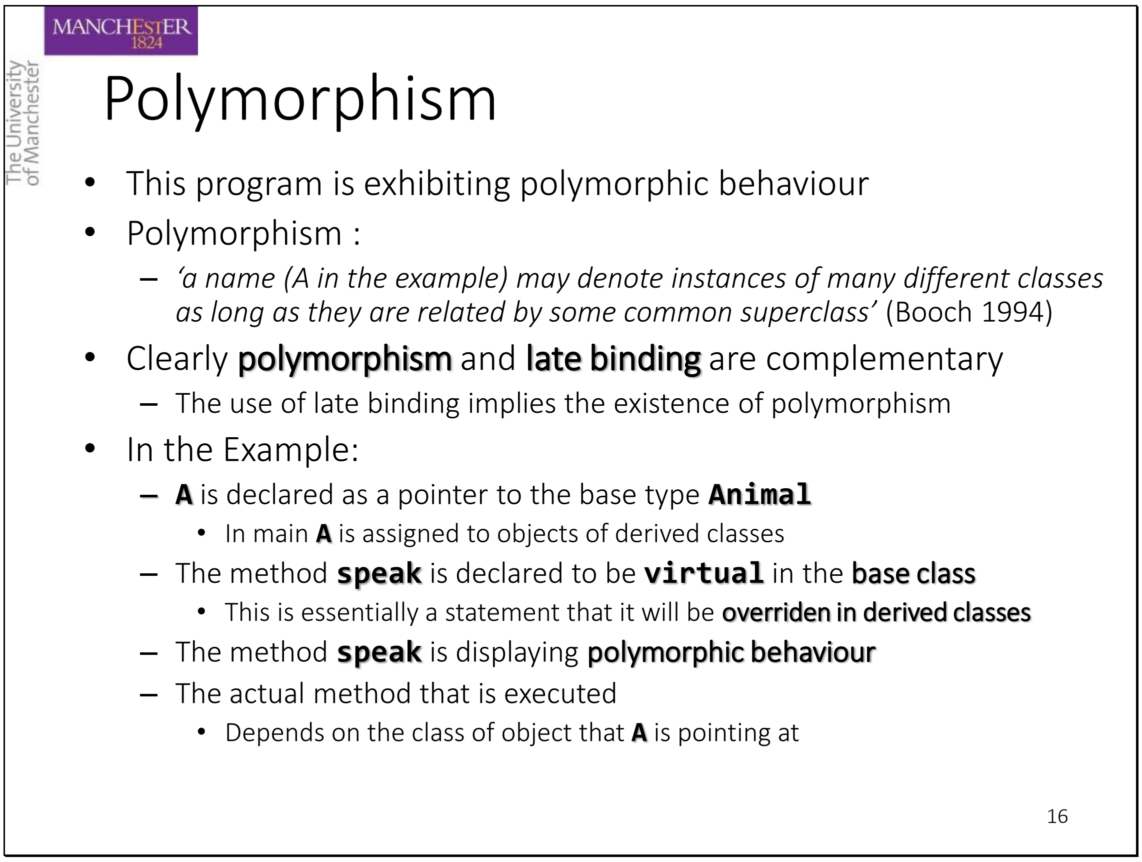
```
void main(void) {
  Animal *A;
  Dog d;
  Cat c;
  Fish f;

  A = &d;
  A->speak; // outputs woof woof
  A = &c;
  A->speak; // outputs miaow
  A = &f;
  A->speak; // outputs blub blub
}
```

15

This example illustrates **late binding**. Although the class of **A** is defined to be **Animal**, the member function that is invoked is determined at run-time on the basis of the dynamic type of the object stored in **A.**

Although it is possible to determine the above binding at compile-time, imagine what would happen if we had a linked list of **Animals**. Using late binding we could store **Animals**, **Dogs**, **Cats** and **Fish** on the list, and invoke the appropriate behaviour. Such a binding can only be done at run-time.

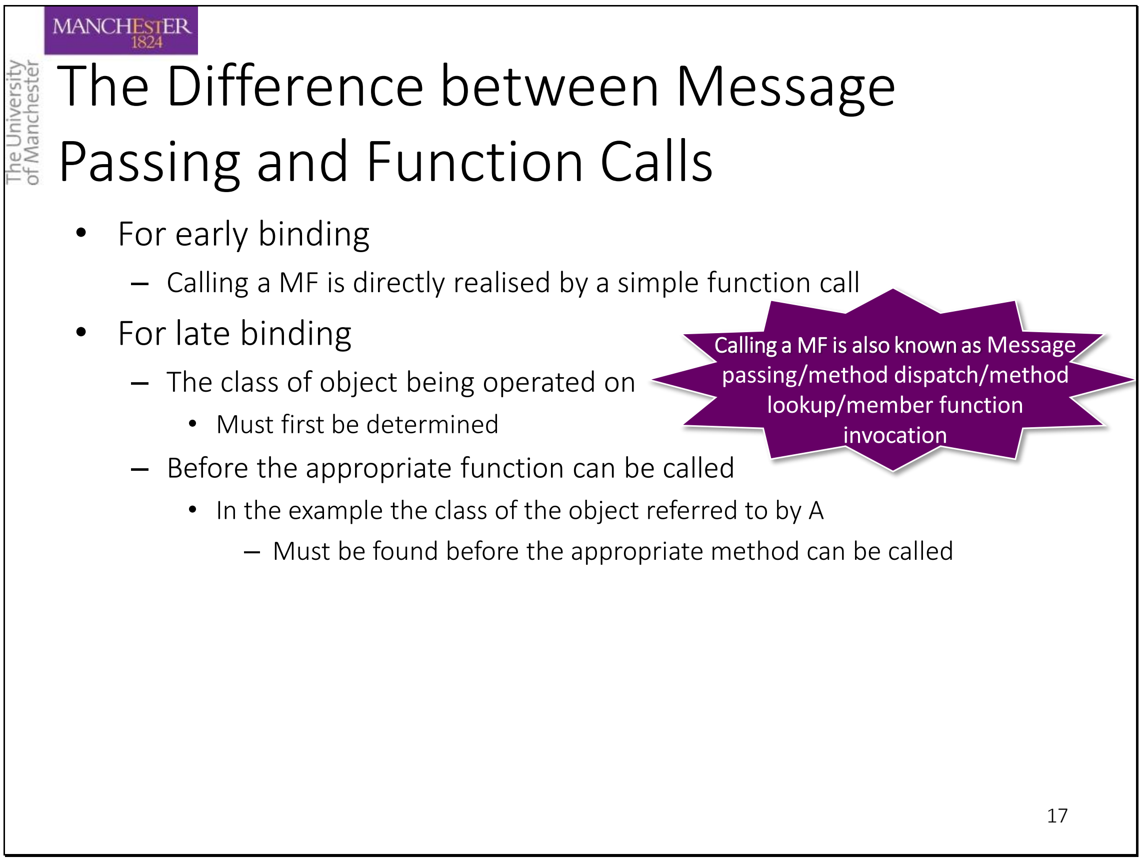
Note that the syntax virtual void **speak(void) = 0;** will be explained shortly.



Polymorphism

- This program is exhibiting polymorphic behaviour
- Polymorphism :
  - *'a name (A in the example) may denote instances of many different classes as long as they are related by some common superclass'* (Booch 1994)
- Clearly **polymorphism** and **late binding** are complementary
  - The use of late binding implies the existence of polymorphism
- In the Example:
  - **A** is declared as a pointer to the base type **`Animal`**
    - In main **A** is assigned to objects of derived classes
  - The method **`speak`** is declared to be **`virtual`** in the **base class**
    - This is essentially a statement that it will be **overriden in derived classes**
  - The method **`speak`** is displaying **polymorphic behaviour**
  - The actual method that is executed
    - Depends on the class of object that **A** is pointing at

16

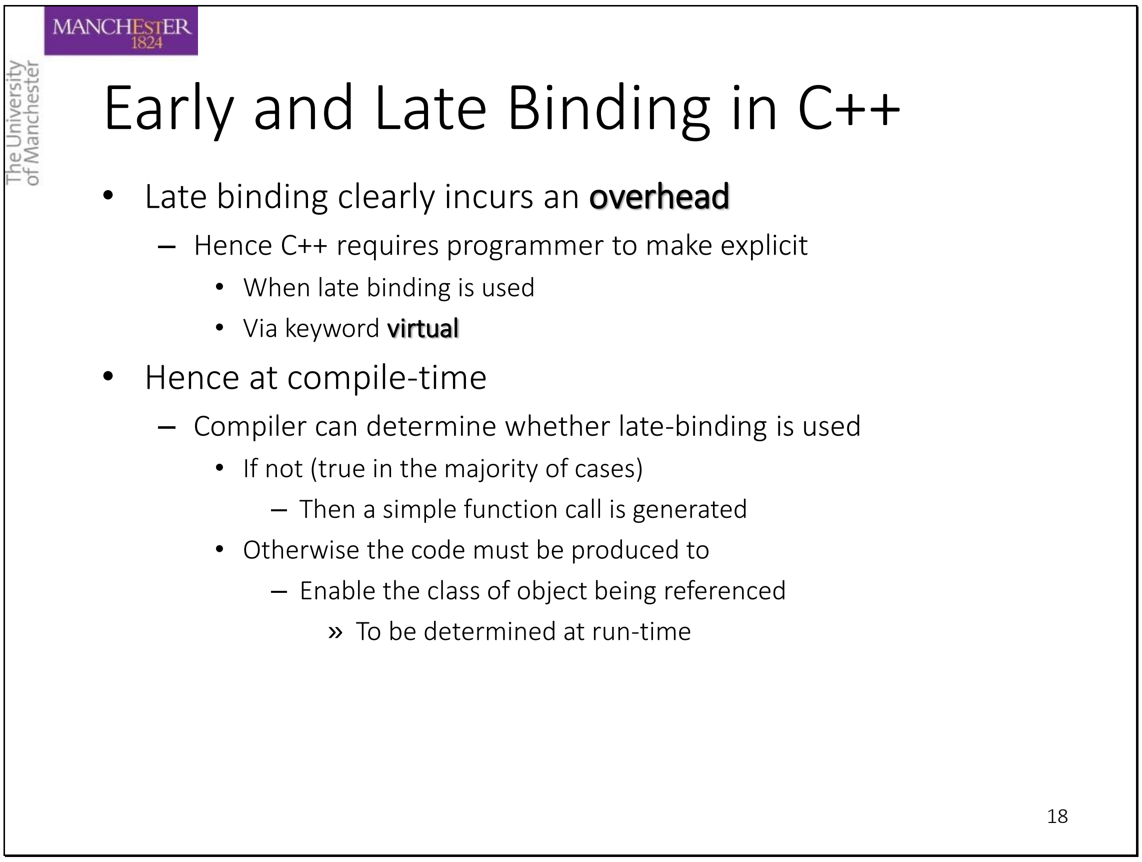


The Difference between Message Passing and Function Calls

- For early binding
  - Calling a MF is directly realised by a simple function call
- For late binding
  - The class of object being operated on
    - Must first be determined
  - Before the appropriate function can be called
    - In the example the class of the object referred to by A
      - Must be found before the appropriate method can be called

Calling a MF is also known as Message passing/method dispatch/method lookup/member function invocation

17

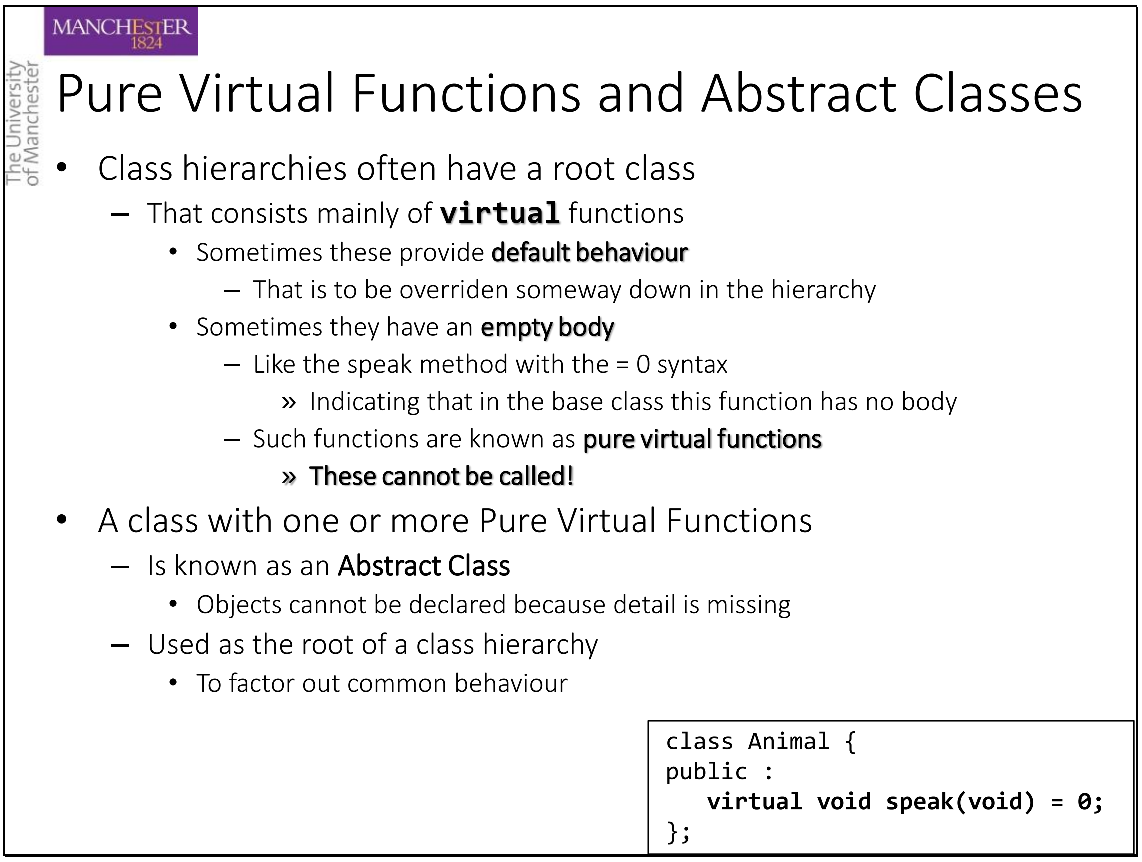


Early and Late Binding in C++

- Late binding clearly incurs an **overhead**
  - Hence C++ requires programmer to make explicit
    - When late binding is used
    - Via keyword **virtual**
- Hence at compile-time
  - Compiler can determine whether late-binding is used
    - If not (true in the majority of cases)
      - Then a simple function call is generated
    - Otherwise the code must be produced to
      - Enable the class of object being referenced
        - » To be determined at run-time

18

It has been observed several times in the unit that message passing is rather more general than a simple function call and this explains why. If **static binding** is used, then message passing is a simple function call. However, if **dynamic binding** is used then the process is more complicated, since the class of the ‘called’ object must be determined and the appropriate method found, and then invoked. This process is called *method lookup*.



Pure Virtual Functions and Abstract Classes

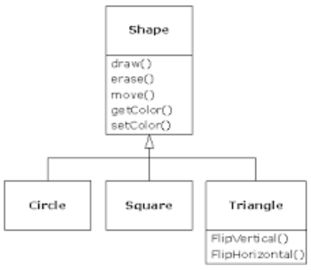
- Class hierarchies often have a root class
  - That consists mainly of **`virtual`** functions
    - Sometimes these provide **default behaviour**
      - That is to be overriden someway down in the hierarchy
    - Sometimes they have an **empty body**
      - Like the speak method with the = 0 syntax
        - » Indicating that in the base class this function has no body
      - Such functions are known as **pure virtual functions**
        - » **These cannot be called!**
- A class with one or more Pure Virtual Functions
  - Is known as an Abstract Class
    - Objects cannot be declared because detail is missing
  - Used as the root of a class hierarchy
    - To factor out common behaviour

```
class Animal {
public :
   virtual void speak(void) = 0;
};
```

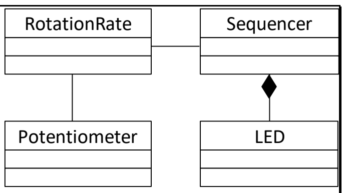
Based on the above, `Animal` is an abstract class, since although it is defined to have a `speak` method, the code body for this cannot be given since specific types of `Animal` (derived classes) ‘`speak`’ in different ways. Hence the ‘ `= 0`’ syntax.

Object Oriented Design

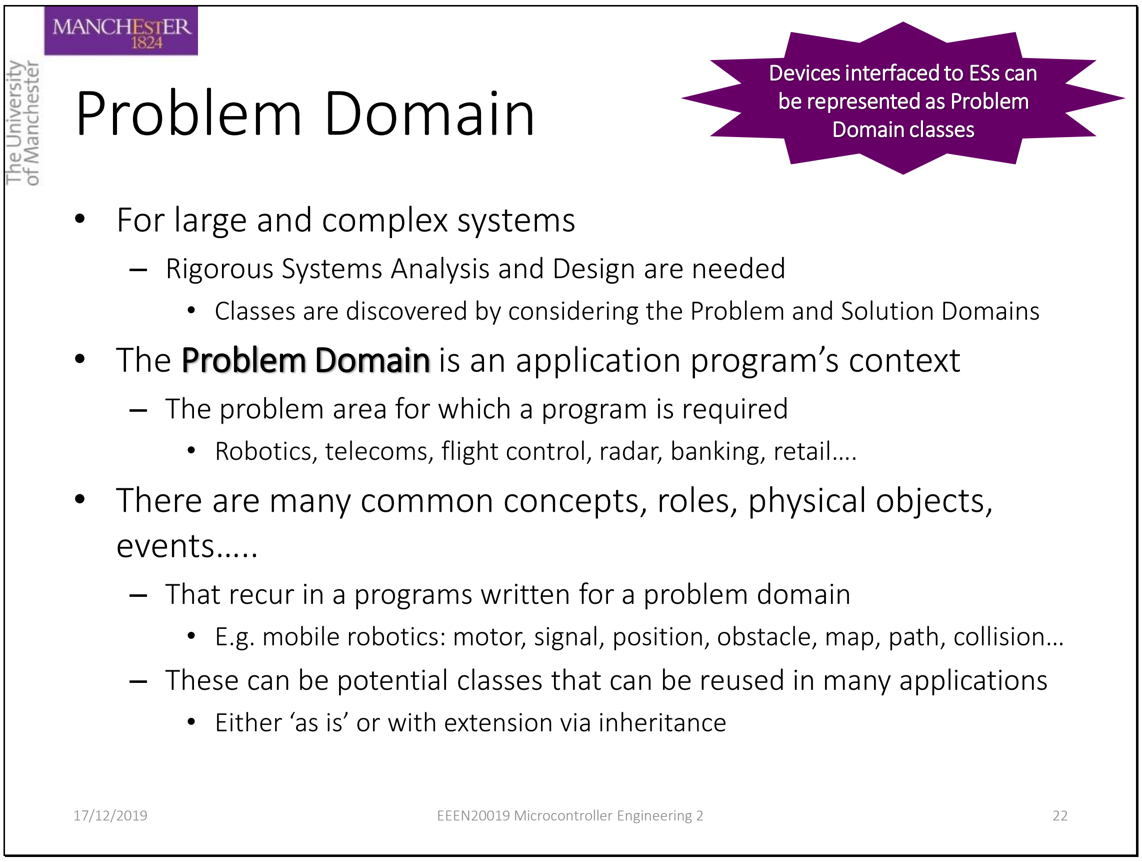
- This unit is NOT directly concerned with Object Oriented Design
  - (or Analysis)
- But, it is helpful to offer guidelines on
  - What makes a good design
  - What makes a good class
  - How the classes needed in a program are identified
- More of this in the ESP lectures
  - On Analysis/Specification and Design



Identifying Classes



- This is an important part of OO software development
- For small programs it is relatively simple
  - E.g. in simple ESs, devices that a program uses
    - Suggest a set of classes that are needed in the program
  - Data structures like buffers, stacks, sets etc
    - Are another group of important classes
  - Classes that coordinate the operation of objects of other classes
    - Controllers or managers are often needed
      - E.g. the **`Sequencer`** class in Example Program 3
- The above sets of classes are **not** the only ones need
  - It is necessary to invent classes to provide program-specific logic and data
    - These may be re-used or may be one-offs
      - E.g. the **`RotationRate`** class in Example Program 3



MANCHESTER 1824

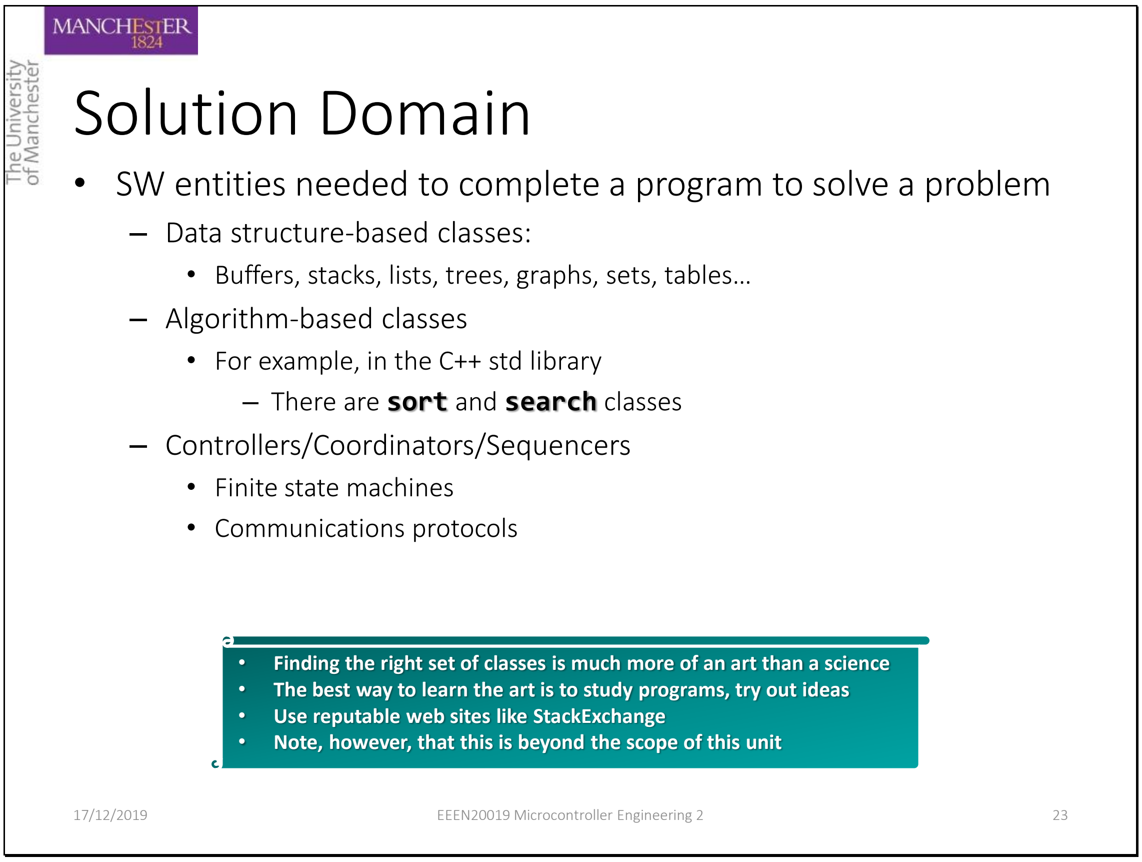
The University of Manchester

Problem Domain

Devices interfaced to ESs can be represented as Problem Domain classes

- For large and complex systems
  - Rigorous Systems Analysis and Design are needed
    - Classes are discovered by considering the Problem and Solution Domains
- The **Problem Domain** is an application program's context
  - The problem area for which a program is required
    - Robotics, telecoms, flight control, radar, banking, retail….
- There are many common concepts, roles, physical objects, events…..
  - That recur in a programs written for a problem domain
    - E.g. mobile robotics: motor, signal, position, obstacle, map, path, collision…
  - These can be potential classes that can be reused in many applications
    - Either 'as is' or with extension via inheritance

17/12/2019 EEEN20019 Microcontroller Engineering 2 22



MANCHESTER 1824

The University of Manchester

Solution Domain

- SW entities needed to complete a program to solve a problem
  - Data structure-based classes:
    - Buffers, stacks, lists, trees, graphs, sets, tables…
  - Algorithm-based classes
    - For example, in the C++ std library
      - There are **sort** and **search** classes
  - Controllers/Coordinators/Sequencers
    - Finite state machines
    - Communications protocols

- **Finding the right set of classes is much more of an art than a science**
- **The best way to learn the art is to study programs, try out ideas**
- **Use reputable web sites like StackExchange**
- **Note, however, that this is beyond the scope of this unit**

17/12/2019 EEEN20019 Microcontroller Engineering 2 23

MANCHESTER 1824

The University of Manchester

Next Steps

- The class identification process yields a set of '**candidate classes**'
  - Not all may be needed in a program
  - Sometimes several identified classes turn out to be the same class
    - But simply with different names
  - Some classes may not really be classes at all
- The message is:
  - Once a set of candidates has been identified, it should be analysed rigorously
- After this evaluation, a set of classes will remain
  - The next steps are to:
    - At a high level, define what the classes do, and what info they must maintain
      - These are termed the **responsibilities** of the class
        - » Eventually these will be refines into a set of MFs and DMs
    - Identify the relationships between the classes
      - Association, composition, aggregation, inheritance

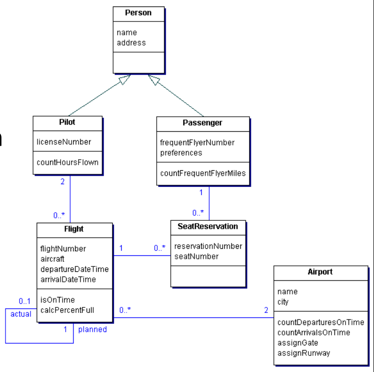
MANCHESTER 1824

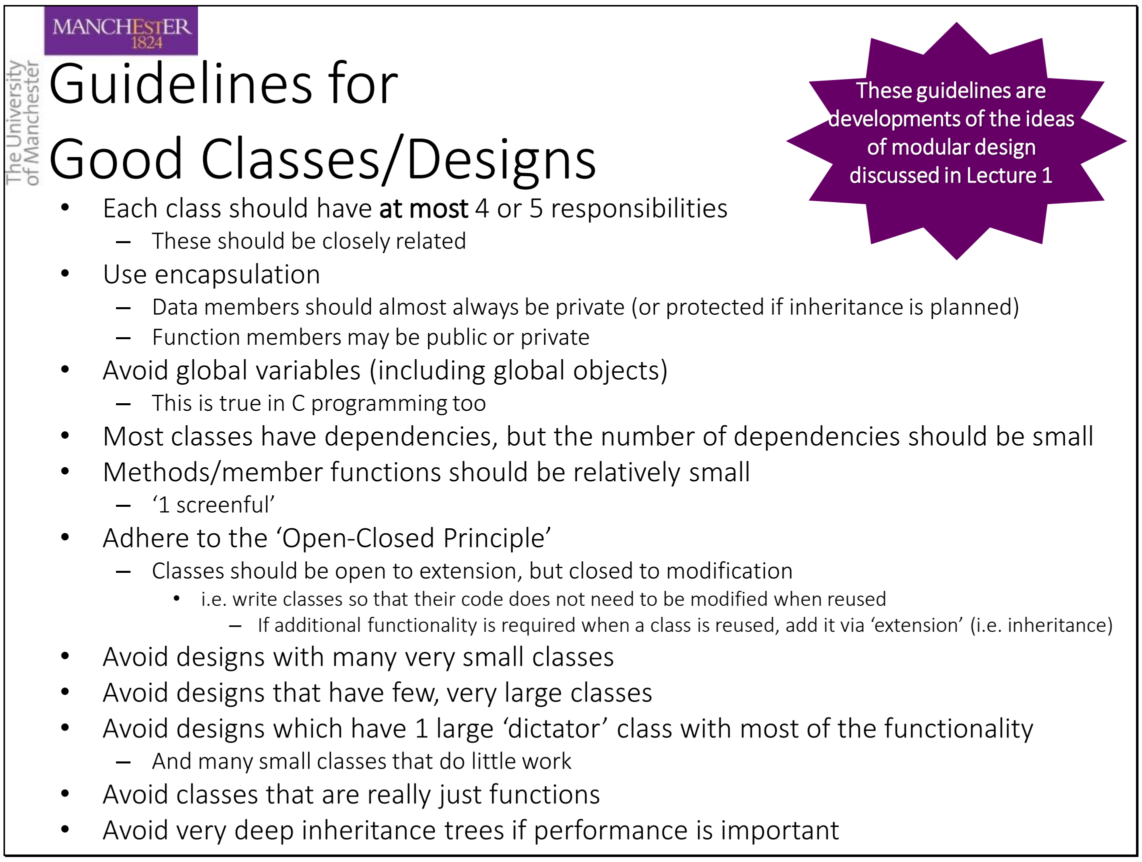
The University of Manchester

Classes

Reminder

- The Class is the basic unit of decomposition in an OO design
  - Embodies one abstraction in problem or solution domain
  - Has a small number of well-defined responsibilities
    - Responsibilities are a high level view
      - of what objects of the class do
      - or the information they are expected to maintain
- Separates interface from implementation
  - Interface defines what objects of the class
    - Can be asked to do
    - The way the class achieves this should be hidden
- Simple and extendible
- Generic as possible





The University of Manchester

MANCHESTER 1824

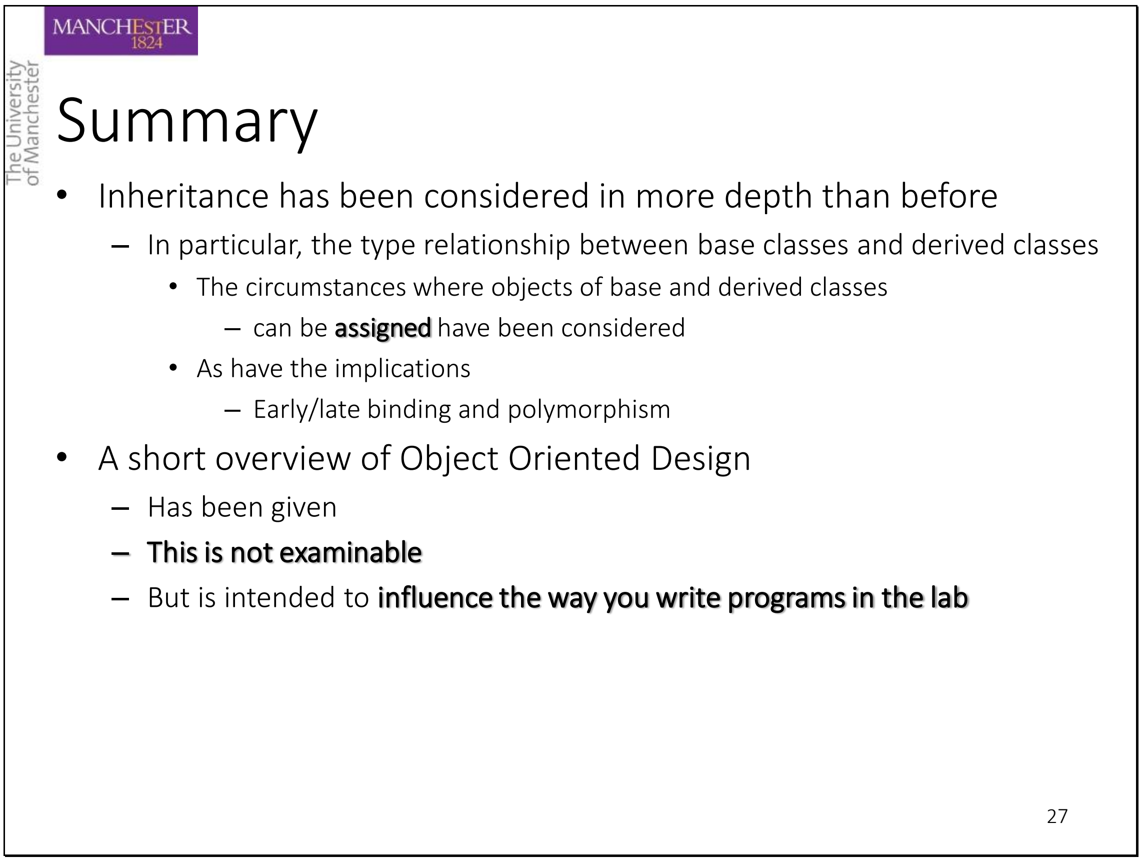
Guidelines for Good Classes/Designs

These guidelines are developments of the ideas of modular design discussed in Lecture 1

- Each class should have **at most** 4 or 5 responsibilities
  - These should be closely related
- Use encapsulation
  - Data members should almost always be private (or protected if inheritance is planned)
  - Function members may be public or private
- Avoid global variables (including global objects)
  - This is true in C programming too
- Most classes have dependencies, but the number of dependencies should be small
- Methods/member functions should be relatively small
  - '1 screenful'
- Adhere to the 'Open-Closed Principle'
  - Classes should be open to extension, but closed to modification
    - i.e. write classes so that their code does not need to be modified when reused
      - If additional functionality is required when a class is reused, add it via 'extension' (i.e. inheritance)
- Avoid designs with many very small classes
- Avoid designs that have few, very large classes
- Avoid designs which have 1 large 'dictator' class with most of the functionality
  - And many small classes that do little work
- Avoid classes that are really just functions
- Avoid very deep inheritance trees if performance is important

One of the issues with software development is that it is not based on laws of nature. Consequently, what constitutes 'good' software is not clear cut. However, experience accumulated over many years has led the software development community to certain views of what is good and bad.

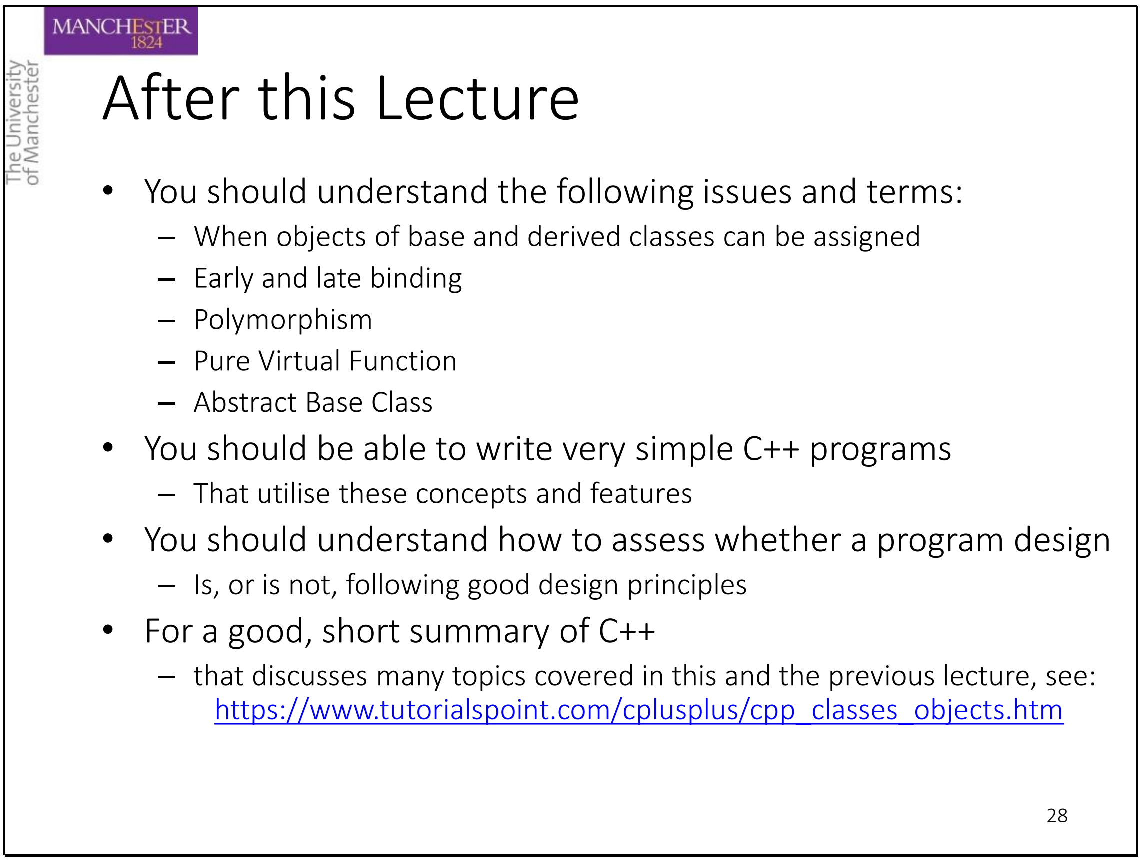
On this basis, the above are **guidelines** – they are **NOT rules**. This means that it is sometimes OK to break these rules, IF THERE IS A GOOD REASON FOR DOING SO. However, if your program design breaks any rules, you should question yourself as to whether you design needs to be improved.



Summary

- Inheritance has been considered in more depth than before
  - In particular, the type relationship between base classes and derived classes
    - The circumstances where objects of base and derived classes
      - can be **assigned** have been considered
    - As have the implications
      - Early/late binding and polymorphism
- A short overview of Object Oriented Design
  - Has been given
  - **This is not examinable**
  - But is intended to **influence the way you write programs in the lab**

27



After this Lecture

- You should understand the following issues and terms:
  - When objects of base and derived classes can be assigned
  - Early and late binding
  - Polymorphism
  - Pure Virtual Function
  - Abstract Base Class
- You should be able to write very simple C++ programs
  - That utilise these concepts and features
- You should understand how to assess whether a program design
  - Is, or is not, following good design principles
- For a good, short summary of C++
  - that discusses many topics covered in this and the previous lecture, see: https://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm

28