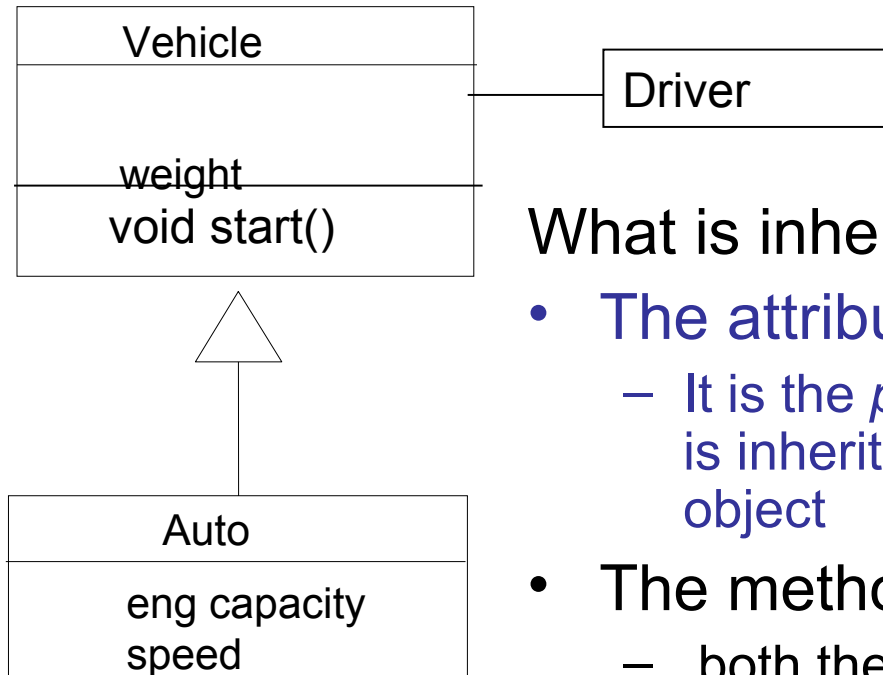


Inheritance

See Savitch
chapter 14 and chapter 15

Inheritance is a class thing!



What is inherited?

- The attributes of a class
 - It is the *presence* of an attribute in a class that is inherited, *not the value* of an attribute in an object
- The methods of a class
 - both their existence and their implementation
- Association is also inherited:
 - *this one is often forgotten*
 - In the example here, an **Auto** object *is a* **Vehicle**, and therefore has an association with a **Driver**
- Constructors ARE NOT inherited!!!

Purpose of using inheritance

- For specialisation
 - To re-use an existing class in the construction of a more specialised one
 - To specialise a class we do not even need access to the source code of the super-class
 - Only do this if you can say “every object of this specialised class is also a member of the general class”
- For generalisation
 - To avoid coding the common features many times in the different classes
 - If we have a set of related classes, gather up all the common features and functions into one super class
 - Should still be able to say “each object of a specialised class is a member of the general class”
- Remember: an object of the derived class has more than one type
 - if it is an Auto, then it is also a Vehicle (because Auto inherits from Vehicle)
 - You must always respect this in designing a class hierarchy

C++ syntax

```
class Vehicle
{
public:
    Vehicle(int)
    int getWeight() const;
    void setWeight(int);
private:
    int weight;
};
```

```
class Auto : public Vehicle
{
public:
    Auto(int, int);
    int getMomentum();
private:
    int speed;
};
```

Terminology:

Base class = super-class

Derived class = sub-class

- Beware: because `weight` is private, then although it exists in the sub-class `Auto`, **it is hidden and cannot be accessed in any methods of the `Auto` class**

- The methods in the sub-class `Auto` must call the `getWeight()` method in order to access the weight:

```
int Auto::getMomentum()
{
    // Momentum is speed × weight
    return speed * getWeight();
}
```

- In order for weight to be directly available to the sub-class, we will need a new access specifier **protected**.

Protected data members

By making the weight of the Vehicle protected

- It is available to the methods of the sub-class Auto
- It is 'as good as private' to any code which is not in a method of Vehicle class or of the derived class Auto

For example, supposing the new method `getMomentum()` in the Auto class is to return the weight times the speed of an Auto.

If weight were private in the Vehicle class:

```
int Auto::getMomentum()  
{  
    return speed * getWeight();  
}
```

Must use the public method to access the private weight

Can use weight directly in the subclass as it's protected rather than private

```
class Vehicle  
{  
public:  
    Vehicle(int);  
    int getWeight() const;  
    void setWeight(int);  
protected:  
    int weight;  
};
```

```
class Auto : public Vehicle  
{  
public:  
    Auto(int, int);  
    int getMomentum();  
private:  
    int speed;  
};
```

But when weight is protected:

```
int Auto::getMomentum()  
{  
    return speed * weight;  
}
```

The public derivation of a class

```
class Vehicle
{
public:
    Vehicle(int)
    int getWeight() const;
    void setWeight(int);
private:
    int weight;
};
```

```
class Auto : public Vehicle
{
public:
    Auto(int, int)
private:
    int speed;
};
```

The `public` derivation means

Anything **public** in the base class is **public** in the derived class

Anything **protected** in the base class is still **protected** in the derived class

Anything **private** in the base class is present and **private** *but hidden* in all derived classes

It is also possible to have **protected** derivation or **private** derivation. These are more restrictive, and not

usually necessary

The 'slicing problem'

A derived class object can be assigned to a base class variable, but the extra data in the derived class will be 'sliced off'

```
class Pet
{
public:
    void print();
    string name;
};
```

We have made `name` and `breed` public for simplicity. In a real application they should be private, and accessed via member functions

```
class Dog: public Pet
{
public:
    void print();
    string breed;
};
```

```
Dog vdog;
Pet vpet;
vdog.name = "Tiny";
vdog.breed = "Great Dane";
vpet = vdog;
vpet.print() //calls print in
             //Pet class
```

- **vpet only stores the name – it does not have a data field to store the breed: breed is sliced off (ie: loss of precision)**

- If a member function is called on `vpet`, it must be a member function of `Pet` class (not added in `Dog`) and even if that member function was re-defined in the `Dog` class, it is the version in the `Pet` class that will be used.
- *Just as well - member functions implemented in `Dog` may include references to the sliced off data!*

Constructors of derived classes

- The first thing to happen in the constructor of a derived class is that **the *default constructor* of the super class is called.**
 - No code is written for this to happen, it happens automatically
 - If no default constructor is available for the base class, there will be a compile error

```
Auto::Auto(int wt, int sp)
: speed(sp)
{
    setWeight(wt); //line 1
}
```

← **Danger!!! This will not compile if there is no default constructor in the Vehicle class**

The code in the Vehicle default constructor executes *before* the code in line 1.

It probably initialises the weight to some default value, which can then be reset as it is here in line 1

Base class Initialisation

```
Auto::Auto(int wt, int sp)
: speed(sp)
{
    setWeight(wt); //line 1
}
```

the default constructor in the `Vehicle` class executes, followed by line 1 to set the weight.

But even if `Vehicle` has a default constructor, we would prefer to use the overloaded constructor `Vehicle(int wt)` for efficiency

We can do this with **base class initialisation**, which tells the compiler to use the overloaded `Vehicle` constructor (instead of the default one) as the first code to be executed in the `Auto` constructor

```
Auto::Auto(int wt, int sp)
: Vehicle(wt), speed(sp)
{
}
```

Now code is more efficient, plus there's no problem even if there isn't a default constructor in the vehicle class

Re-defining member functions

- Suppose we have a vehicle classification system which includes `Truck`, a sub-class of `Auto`. Trucks consist of two parts, the front engine which pulls a trailer. Both the front engine and the trailer have their own weights.

```
class Truck: public Auto
{
    public:
        Truck();
        Truck(int, int, int);
        int getWeight() const;
    private:
        int trailerwt;
};
```

- The `getWeight()` member function inherited from `Auto` will return the engine weight only.
- We can re-define this member function so that when it is called **On a `Truck` object** it will return the combined weight
- It must be included in the `Truck` class list of member functions, and an implementation must be provided

```
int Truck::getWeight() const
{
    return (Vehicle::getWeight() + trailerwt);
}
```

*Note **scope resolution operator** used to access the `Auto` `getWeight()` method*

Calling the methods of a derived class and its super-class

- The next example shows the actual use of the member functions of the class Truck to display several weights.

```
int main()
{
    Truck lorry (3000, 120, 2500);
    cout << "\nFront weight: "
         << lorry.Vehicle::getWeight()
         << "\nFront + trailer weight: "
         << lorry.getWeight()
         << endl;
    return 0;
}
```

Note use of the scope resolution operator here to get the weight of the front part of lorry only

Whereas this calls the getWeight method of the Truck class, and provides total of front and back weight.

- If we also had an overloaded setWeight(int, int) member function in the Truck class, to set the 2 parts of the weight, then this over-ridden 2-arg setWeight would hide the single arg setWeight in the Vehicle class.
 - Only the 2-arg version could be called on a Truck object (unless the scope resolution operator is used)

```
lorry.setWeight(2300, 5000); // sets the 2 parts of the weight
lorry.Vehicle::setWeight(4000); // sets the weight of the front part only
```

Where does polymorphism come in?

- Polymorphism is the ability to decide dynamically (at run time) which function code is appropriate for a particular method call.

```
Truck aTruck;
```

```
Double wt = aTruck.getWeight();
```

- the re-defined code for a Truck is only executed because we call the method on a Truck and not an Auto. It is decided at compile time - **NOT POLYMORPHISM!**

- However, a Truck is an Auto, so we could store it in an Auto variable - will this force polymorphism to be used?

```
Auto myAuto = aTruck;
```

- Note that we have lost precision here: *only the data members of the truck which it inherited from Auto* are stored in the `myAuto` variable. (this is the slicing problem)
- `myAuto.getWeight()` will execute the `getWeight` method in the Auto class! (just as well, as the extra data member `trailerwt` has been sliced off) - **STILL NO POLYMORPHISM!!**

Polymorphism with pointers?

- Suppose we get a little bit cleverer still:

```
Auto *myAuto = new Truck(3000, 120, 2500);
```

- Now we have not lost precision: we are storing a pointer to the whole Truck in a “pointer to an Auto” variable.
- We might think that now we could say
`myAuto->getWeight()` to execute the `getWeight` method in the `Truck` class.
- It would certainly be safe to do (no data has been sliced off)
- But still because the variable is stored *as a pointer to an Auto*, C++ will decide at compile time to use the `Auto getWeight()` method!
- Again, **NO POLYMORPHISM!!!**

We would like to use polymorphism, (aka dynamic or late binding), and have the `getDetails()` method of the `Auto` used in this case, and we CAN tell C++ to do this by using the **virtual** keyword

Polymorphism

- If we label a member function in the base-class as a virtual function, then:
 - At compile time, the compiler doesn't link in the code to be executed.
 - Instead, it sets up the machinery needed for this to happen *at run-time*
 - It only does this when it encounters a *pointer* to an object

The `getWeight()` method is declared virtual at the 'top' of the hierarchy of classes

The method is then virtual in all sub-classes (virtual keyword is not used in sub-classes)

```
class Vehicle
{
public:
    Vehicle(int)
    virtual int getWeight() const;
    ...
private:
    int weight;
};
```

```
Auto *myAuto = new Truck(3000, 120, 2500);
myAuto->getWeight();
```

Now the `getWeight()` method in the `Truck` class will be used even though it is accessed through an `Auto` pointer

Where is polymorphism useful?

1. If a *reference* or *pointer* to the base class is passed as an argument to a function, it should work correctly whichever sub-class the object referred to is an instance of:

```
bool operator< (Employee &emp1, Employee &emp2)
{
    if (emp1.getRemuneration() < emp2.getRemuneration())
        return true;
    else
        return false
}
```

emp1 and emp2 may be references to objects which belong to a sub-class of Employee

2. If an array holds a collection of *pointers* to the base class, the appropriate method should be called for each member of the array
3. [NB array can hold *pointers*, but NOT *references*!]

```
Employee* myemps[20];
...
double total = 0;
for (int i = 0; i < 20; i++)
    total += myemps[i]->getRemuneration();
```

Any array entry may be a pointer to an object which belongs to a sub-class of Employee

Why is polymorphism not automatic?

- The overhead of using a virtual function is high
 - Uses more storage
 - Affects the run-time performance
- So the programmer is given discretion over whether to use them
 - If you expect to need the advantages of polymorphism, then make the function virtual
 - if you do not expect to need the advantages of polymorphism, then don't make the function virtual

How do virtual functions work

- The compiler adds one **virtual function table** for each class which has virtual functions, and for each derived class which re-defines some virtual functions.
 - The virtual function table has an entry for each virtual function
 - At compile time, the correct function code is linked to each entry in the virtual function table for the class
- It also adds an **extra data member** to each object created. This extra data member points to the appropriate virtual function table.
- Similar mechanism is used in all programming languages which support polymorphism

The virtual function table

When the Employee class is compiled, the compiler sees that a virtual function is present in the Employee class. It therefore

- constructs a virtual function table for the Employee class

- adds a pointer to a virtual table as an unseen extra data member of the class

```
class Director : public Employee
{
    ...
    void getRemun() ;
    pointer to vTable
};
```

Since Director class redefines the virtual function, it will have its own vtable

```
class Employee {
    ...
    virtual void getRemun() ;
    pointer to vTable
};
```

vTable for class Employee

getRemun() - linked to Employee method code

+ Entries for any other virtual functions in the Employee class

vTable for class Director

getRemun() - linked to Director method code

+ Entries for any other virtual functions in the Director class

How do virtual functions work - cont

- Whenever a new director object is created, its *vtable pointer* points to the virtual table for the `Director` class
- The pointer to the new Director object could be stored in a pointer to an `Employee`
 - But its *vtable pointer* will still point to the vtable for a `Director`
- When the `getRemun()` function is called on a ***pointer*** to this object, the *vtable pointer* will be followed and the code linked to the `getRemun()` entry in the virtual table for the `Director` class will be executed.
 - This only happens when the `getRemun` function is called on a *pointer* to an `Employee`
 - It doesn't happen if the `Director` object was stored as an *actual* `Employee` object – because precision has been lost, and functions specific to the `Director` may use data which has been sliced off.

Destructors of a derived class

- Recall, destructors do NOT ‘destroy’ the object, their purpose is to ‘tidy up any loose ends’ while the data in the object is still accessible.
- Destructors are called just before the memory allocated for an object is released:
 - For a local object that is when the code block (usually a function) in which they were declared is completed.
 - For an object created using ‘new’, that is only when its memory is released using the delete keyword.
 - For global objects, memory is released when the program completes.
- If the object is of a derived class, then *after* the code in the derived class destructor is executed, the destructor of its base class is called.

Virtual Functions and Destructors

- If a class has any virtual functions, the destructor should also be made virtual.
- This means that the destructor for the derived class will be called even if the derived object is held in a base class pointer
- Suppose some clean up is required in the destructor for the derived class, it would not be called if the base class destructor were not virtual.

Syntax Summary

Declaring a sub-class

```
class Auto : public Vehicle
{
    ...
};
```

Public derivation from
the super-class Vehicle

Initialising the part of an object inherited from a super-class

```
Auto::Auto(int wt, int sp)
:Vehicle(wt), speed(sp)
{...}
```

Weight set by calling the
1-arg constructor of the
Vehicle (base-class
initialisation)

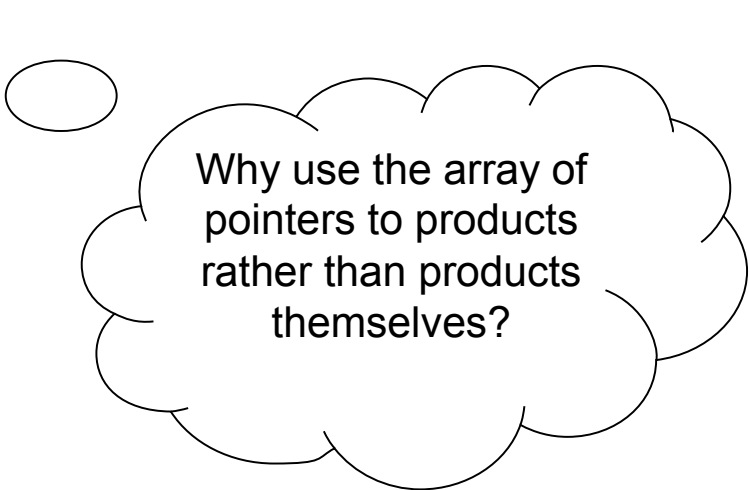
Declaring that a method will be used polymorphically

```
class Vehicle
{
public:
    virtual int getWeight();
    ...
};
```

Use the **virtual**
keyword in the super-
class (NB not repeated
in sub-classes)

Practice

- Write 2 classes, for products of type `Software` and `Book`. The `getPrice()` method of the software product includes VAT at 21%, the `getPrice()` method for the `Book` class does not. Both classes inherit from the super class `Product` which defines the `getPrice()` method as virtual, and returns the price without VAT
- Declare an array of 10 *Pointers* to `Product`.
- Add 10 products to the array, which are a mixture of `Software` and `Book` products.
- Sort the array in ascending order of price.



Why use the array of pointers to products rather than products themselves?