Module 10:
Simple Inheritance

In this module we will cover

- **Inheritance**
- **Method overriding in sub classes**
- **Rules for constructor inheritance**

Aims of this module

The aim of this module is to familiarise students with inheritance

Inheritance allows you to define a class based upon a pre-existing class.

The new class is said to "inherit" all of the public services of the base class. In other words you get all of the methods of the base class free. Inheritance is a key component of OO languages and by the end of this module it is intended that you will be able to write a simple inherited class.

The module also covers the related topic of overriding base class methods.

10.1 Inheritance - basics

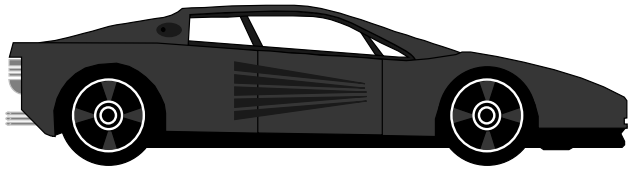
Suppose you have some problem where you identify a class which you need, but

⇒ the class you need is similar to some already existing class

This begs question: why should you have to write the code all over again ?

Consider simple case where all you want to do is add a small amount of functionality to an existing class.

- **Example: The Vehicle class**



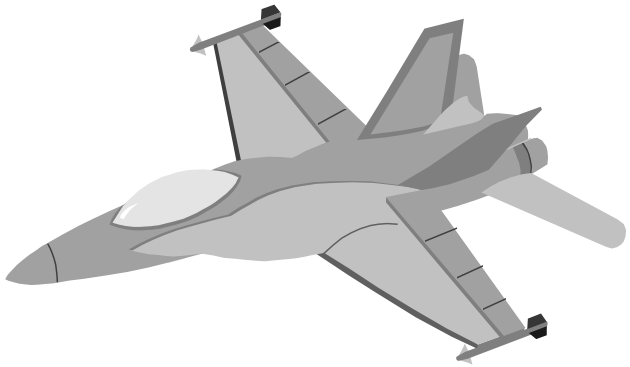
```
class Vehicle
{

  public:

    void stop( ) ;
    void go( float dist ) ;
    void setSpeed( float s ) ;
    void turn( float degrees );

};
```

- All very good - now you can write your playstation game

Need this:

```
class Aircraft
{

  public:

    void stop( ) ;
    void go( float dist );
    void setSpeed( float s ) ;
    void turn( float degrees );

    void rotate( float degrees )

};
```

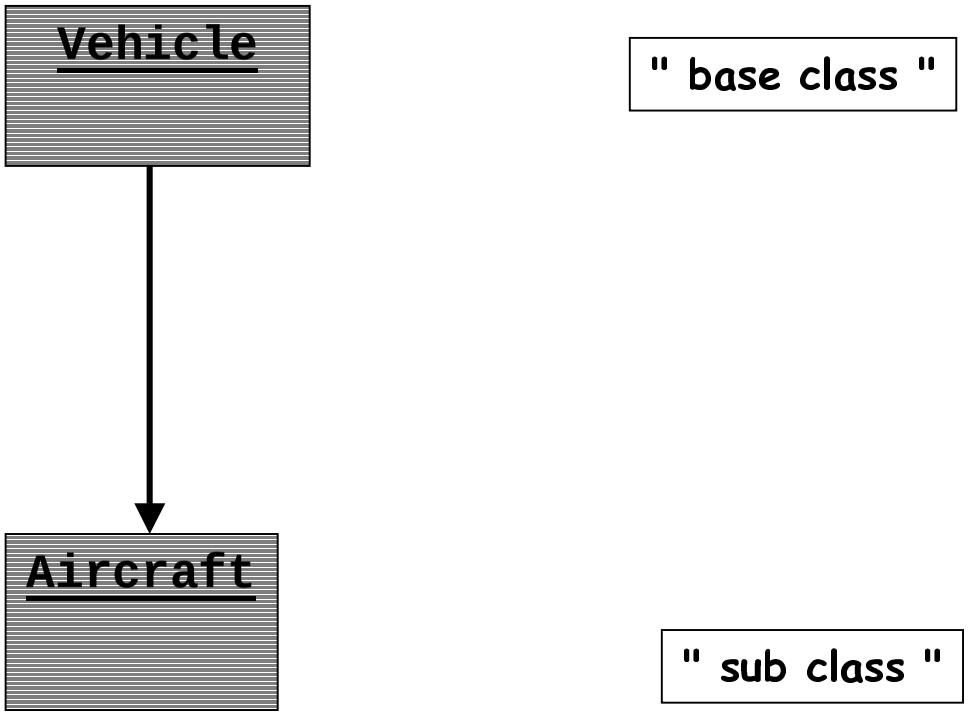
...or can you....



Does this mean we have to re-write a whole new class just to add one method ?

You don't have to re-write the whole thing !

We can make a "sub class" which "inherits" from the "base class "



```
// Example of inheritance in C++

class Aircraft : public Vehicle
{




};
```

This is how you declare one class to inherit from another

The Aircraft class now has available all of the characteristics of Vehicle class free.

In particular:

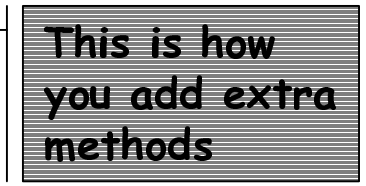
Vehicle class methods

are also

Aircraft class methods.

without having to write a line of code

```
// Example of inheritance in C++

class Aircraft : public Vehicle
{

  public:

    void rotate( float degrees ) ;

  private:

    float rotationAngle ;
    float height ;

};
```



```
// Example of inheritance in C++

class Aircraft : public Vehicle
{

  public:

    void rotate( float degrees ) ;

  private:

    float rotationAngle ;
    float height ;

};
```

This is how you add extra member variables needed for extra characterisation of the object

Now we need to supply the extra method in the normal way,
i.e. in a .cpp file somewhere we add:

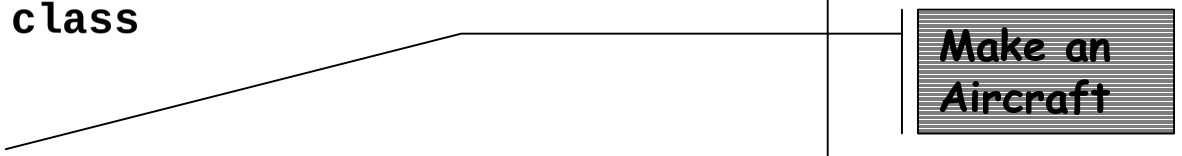
```
// Rotate method for Aircraft

void Aircraft::rotate( float angle )
{

  rotationAngle = angle ;

}
```

..we will worry about height in just a minute..

So we have written an Aircraft class by inheritance

How do you use the the Aircraft class ?

```
// Program fragment to show how you could
// use the Aircraft class

// Make a plane
Aircraft spitfire ;


//Taxi it
spitfire.turn( 45. ) ;        // Turn 45 degrees
spitfire.setSpeed( 30. ) ;    // 30 Km/hr for taxi
spitfire.go( 60. ) ;          // Set in motion


//Stop and prepare for take off
spitfire.stop() ;
spitfire.turn( 30. ) ;        // Turn onto runway
spitfire.setSpeed( 200. )     // Take off speed


// Now take off
spitfire.go( 500. ) ;
spitfire.rotate( 40. ) ;      // Now we leave ground
```



```
// Program fragment to show how you could
// use the Aircraft class

// Make a plane
Aircraft spitfire ;


//Taxi it
spitfire.turn( 45. ) ;        // Turn 45 degrees
spitfire.setSpeed( 30. ) ;    // 30 Km/hr for taxi
spitfire.go( 60. ) ;          // Set in motion


//Stop and prepare for take off
spitfire.stop() ;
spitfire.turn( 30. ) ;        // Turn onto runway
spitfire.setSpeed( 200. )     // Take off speed


// Now take off
spitfire.go( 500. ) ;
spitfire.rotate( 40. ) ;      // Now we leave ground
```

Use methods of the Vehicle class

-

we didnt have to write these

```
// Program fragment to show how you could
// use the Aircraft class

// Make a plane
Aircraft spitfire ;


//Taxi it
spitfire.turn( 45. ) ;        // Turn 45 degrees
spitfire.setSpeed( 30. ) ;    // 30 Km/hr for taxi
spitfire.go( 60. ) ;                // Set in motion


//Stop and prepare for take off
spitfire.stop() ;
spitfire.turn( 30. ) ;        // Turn onto runway
spitfire.setSpeed( 200. )     // Take off speed


// Now take off
spitfire.go( 500. ) ;
spitfire.rotate( 40. ) ;      // Now we leave ground
```

Use the extra method we added

10.2 Over-riding methods in the sub class

In the last example we simply added a bit of functionality to an existing class.

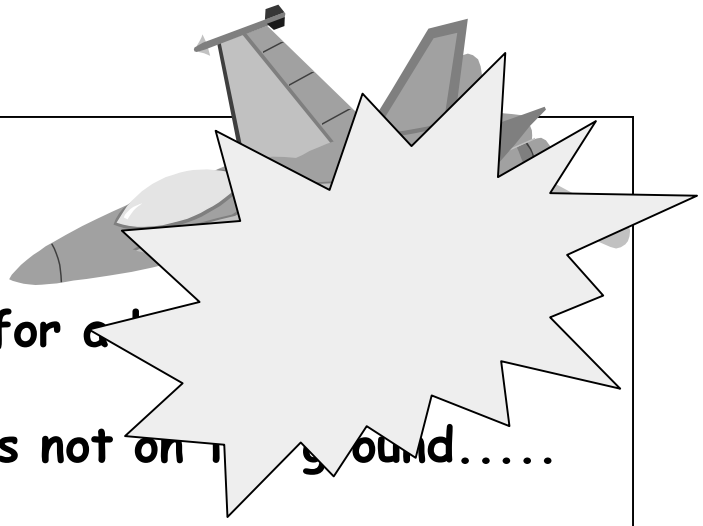
We left the pre-existing methods in the base class alone.

What if you want to modify the operation of one of the base class methods ?

Example:

The concept of `stop( )` is always fine for a

However if you `stop( )` a plane that is not on the ground.....



Another example:

We failed to notice that we also have to modify the `go( )` method to update the height above the ground.

We clearly need to "intercept" any invocation of go( ) and stop( ) and do something else.

Heres how

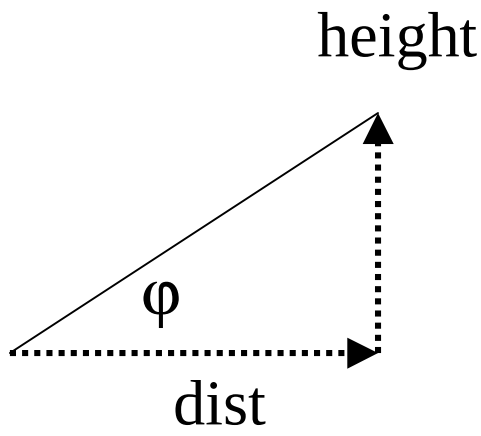
1. First you declare these methods in the new class with exactly the same signature as before.

This indicates that you intend to over-ride these methods.

```
// Example of inheritance in C++

class Aircraft : public Vehicle
{

  public:

    void rotate( float degrees ) ;
    void go( float dist ) ;
    void stop( )

  private:

    float rotationAngle ;
    float height ;

};
```

2. Then you supply the code for these methods in the normal way

```
// Over-riding the go method

void Aircraft::go( float dist )
{

   // First set the height
   height += dist / tan( rotationAngle ) ;

   // Now invoke the Vehicle go method
   Vehicle::go( dist ) ;

}
```



```
// Over-riding the go method

void Aircraft::go( float dist )
{

   // First set the height
   height += dist / tan( rotationAngle ) ;

   // Now invoke the Vehicle go method
   Vehicle::go( dist ) ;

}
```

Here we perform some operations on the new member variables we have added to the class

```
// Over-riding the go method

void Aircraft::go( float dist )
{

   // First set the height
   height += dist / tan( rotationAngle ) ;

   // Now invoke the Vehicle go method
   Vehicle::go( dist ) ;

}
```

This is new !!!!!

This shows how to call a base class method

We use the

"scope resolution operator"

... and the stop method..

```
// Over-riding the stop method

void Aircraft::stop(  )
{

   // Check the height !!!!

   if( height == 0 )
   {
     //safe to stop
     Vehicle::stop( ) ;
   }
   else
   {
     cout << " please land first ! " << endl;
   }
}
```

The material of the last few slides is not trivial to assimilate !

Therefore go over it again now and ask questions if necessary.

Student exercise

Think of two classes which you think might be good candidates for inheritance.

Write a base class with some methods

Write a sub class which inherits from it, and
-adds some new methods
-overrides one method of the base class

Write a main program which makes instances of the sub class.

Demonstrate the use of the inherited methods
(suggest putting cout << statements all over the place to show which methods get called)

If you cant think of your own classes for this exercise then see suggestion on next slide:

Suggested example 1 (simple but boring)

Base class:

Represents an open cylinder (length, radius)

Has methods to return `volume( )` and `area( )`

Derived class:

Represents a closed cylinder

(therefore `area( )` method needs over-riding)

Suggested example 2 (need to know some particle physics here)

Base class:

Represents a ThreeVector

Derived class:

Represents a relativistic FourVector

`mass( )` method needs to be added
`magnitude( )` method needs overriding
`magnitude( )` dotProduct method needs overriding

Suggested example3 (for the confident)

Base class:

```
SuperHero // has: strength, weaponStrength, numberOfLives....
```

Has methods to

```
bool fight( SuperHero& adversary ) // compares strength,weapon
int livesLeft( )  // returns number of lives left
```

Derived classes:

InvisibleSuperHero

Need to override fight method to give advantage

LifeSuckingSuperHero

Need to override fight method to add opponents life if it wins

FlyingSuperHero

Need to add fly() method

Need to override fight method so it cant be attacked in the air

10.3 Rules for constructor inheritance

- **The rules for constructors are complex**
- **You will never remember them all**
- **Even if you do you will forget by the time you need to use them**

⇒ **We will have a quick run through here**

⇒ **Then when you need them I suggest you do what I do...**
....ask an expert friend who you think is smarter than you ...

If both classes have default constructors:

I.e the base class has a constructor which look like this

```
// Base class

class Thing
{
  public:

    Thing(  ) ;
    ....
}
```

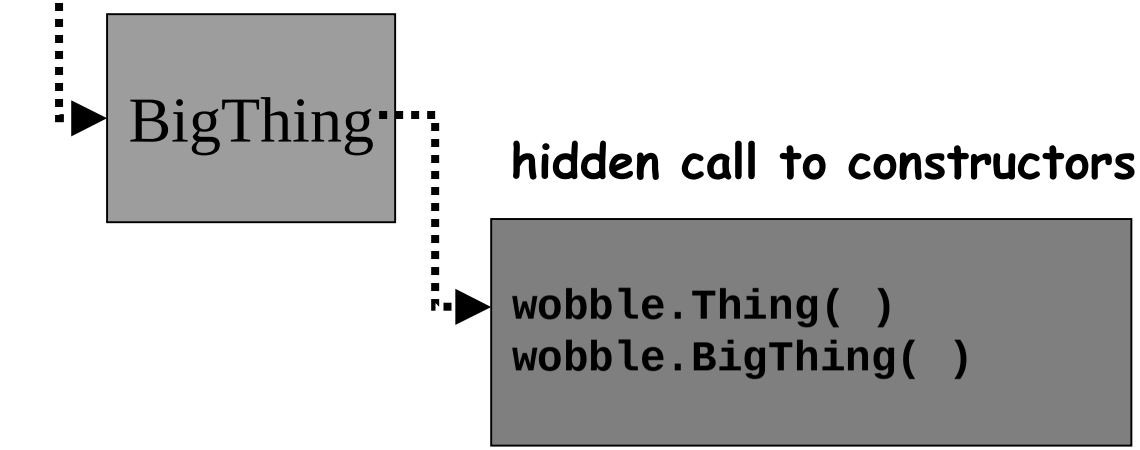
...and the sub class has a constructor which look like this

```
// sub class

class BigThing: public Thing
{
  public:

    BigThing(  ) ;
    ....
}
```

Then when you make a `BigThing` with no constructor arguments:

⇒ **First the default constructor `Thing()` gets invoked**
⇒ **Then the default constructor `BigThing()` gets invoked**

```
// Make a big thing using
// default constructor

BigThing  wobble;

....
....
```



If both classes have both default and non trivial constructors:

I.e the base class constructors which look like this

```
// Base class

class Thing
{
  public:

    Thing( ) ;
    Thing( int a, int b ) ;
}
```

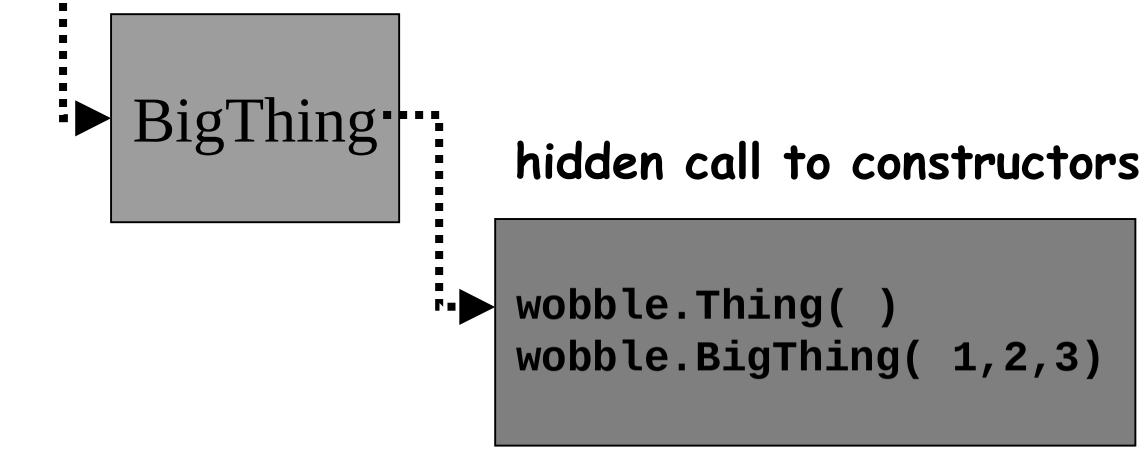
...and the sub class has constructors which look like this

```
// sub class

class BigThing: public Thing
{
  public:

    BigThing( ) ;
    BigThing( int a, int b, int c) ;
}
```

Then when you make a `BigThing` with arguments `( a, b, c )`:

⇒ **First the default constructor `Thing()` gets invoked**
⇒ **Then the constructor `BigThing ( a, b, c )` gets invoked**

```
// Make a big thing using
// non-trivial constructor

BigThing wobble(1, 2, 3);

....
....
```



This is almost certainly NOT what you want.

It is more likely that you want to invoke the not-trivial constructor `Thing( int a, int b)`

To do this you have to put in a specific directive to do this in the `BigThing` constructor:

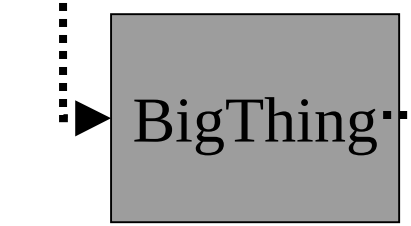
Like this:

```
// Non-trivial Constructor for BigThing

BigThing::BigThing( int a, int b, int c )
  : Thing( a, b )
{
....
....
}
```

Then when you make a `BigThing` with arguments `( a, b, c )`:

⇒ First the constructor `Thing( a, b )` gets invoked
⇒ Then the constructor `BigThing ( a, b, c )` gets invoked

```
// Make a big thing using
// non-trivial constructor

BigThing wobble(1, 2, 3);

....
....
```



hidden call to constructors

```
wobble.Thing( 1, 3 )
wobble.BigThing( 1,2,3)
```

Think about this.

You have not seen anything like this before !

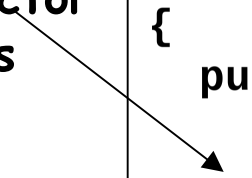
The place where the constructor for Thing has been invoked

```
: Thing( a, b )
```

is NOT in the body of any function !

Corollary :
If the base class does not have a default constructor

I.e the base class only has a constructor which look like this



```
// Base class

class Thing
{
  public:

    Thing( int a, int b ) ;
    ....
}
```

Then you MUST write constructor(s) for BigThing which explicitly invoke the Thing(a, b) constructor.

This will fail because the compiler will tell you that it cant find a default constructor for Thing( ) to invoke

```
// Bad constructor for BigThing

BigThing::BigThing( .... )
{
....
}
```

This will work because you tell the compiler which Thing constructor to invoke

```
// Good constructor for BigThing

BigThing::BigThing( ... )
: Thing( ... two integers in here.. )
{
....
}
```

Student exercise for your own time if you want to get this straight

Think of two classes which you think might be good candidates for inheritance.

Write a base class with a default constructor and at least one non trivial constructor

Write a sub class which inherits from it, and has a default constructors as well as at least one non-trivial constructor.

Write a main program which makes instances of the sub class.

Play around with the invocation of base class constructors as described on the previous slides. [Put `cout <<` statements in all constructors so that you can see which one is called]

Summary of Module 10: Inheritance

In this module we have covered the following topics.

- Basics of inheritance
  - class child : public parent
  - adding of methods
  - adding new member variables
  - use of inherited methods
- Over-riding base class functionality
  - over-riding a method of the base class
  - calling the base class method explicitly with the scope resolution operator
- Rules for constructor inheritance
  - default is that base class default constructor gets called
  - for any other action you must explicitly call the base class constructor you want