Dynamic Polymorphism

Overloading Method Names

- With class hierarchies methods of the base class are inherited by the derived classes.
- For example a method defined in the base class may be called by any object of any derived class.
- However this is very restrictive if we can only use inherited methods in our derived classes
- For example our derived class may use a method from the base class which is not aware of attributes in the derived class.
  - Either re-write a new method in the derived class (using the same method name which is known as overriding)
  - Or extend the method in the derived class by first calling the base class method and then executing the derived class method code

Abstract Methods (1)

- It may be that two derived objects methods do not have much in common, but need the same name as their functionality are the same.
- In this case it makes no sense to define the method in the base class as the implementation will be different
- However this in not good practice as having overloaded methods names in different parts of the classification hierarchy without common root will mean that the method will be applied in an ’ad-hoc’ way.

Abstract Methods (2)

- This may not seem like an issue now but when we look at the use of dynamic objects it may cause problems
- To overcome this, we use the method name in the base class as a sort of place holder to allow the correct sequence of methods to be called.
- This method will not have any functionality or implementation, it just sits there and does nothing.

C++ Syntax

```
class BaseClass
{
private :
  int x;
public :
  void setX(int xin)
  {
    x=xin;
  }
  void showX()
  {
    cout<<"Base x ="<<x<<endl;
  }
};
```

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

    void getY()
    {
      cout<< "Base y ="<<y<<endl;
    }
};
```

Simple Program

```
#include <iostream>
using namespace std;
#include "baseclass.h"
int main(void)
{
BaseClass base_object;
DerivedClass derived_object;
base_object.setX(7);
derived_object.setX(12);
base_object.showX();
derived_object.showX();
}
```

- **Program output**

  base x = 7

  base x = 12

What is wrong with this output ?

Overriding the derived Method

- The output from the derived class is incorrect as it says **`base x =12`** where it should say **`derived x = 12`**
- To modify this output we need to override the showX method in the DerivedClass to do this we add the method definition to the DerivedClass

```
class DerivedClass:public BaseClass
{
private :
    int y;
public :
    void setY(int yin)
    {
      y=yin;
    }
    void getY()
    {
      cout<<"Base y ="<<y< <endl;
    }
    void showX()
    {
      cout<<"Derived X ="<<x<<endl;
    }
};
```

Defining abstract Methods (1)

- In the previous example the a method is overridden in the derived class

- In the first example program the compiler was able to bind a correct method from the base class into the derived class for the **`showX`** method

- In the second example class the new **`showX`** method will be used as the compiler finds this first.

Defining abstract Methods (2)

- However if the method is 'abstract' in the base class, then there will not be any implementation of that method inherited.
- An abstract method exists purely for the purpose of being overridden by methods in the derived classes.
- So in contrast to other methods which may be used throughout a class's descendants, abstract methods must be overridden by all inheriting classes to provide a useful behavior

Extending Inherited methods

- In the previous example we applied polymorphism to override (replace) an inherited method definition
- We may also, if we wish, simply extend the definition of an inherited method by calling it in a derived class method and add in extra implementation detail.
- The following class definition shows a simple class (ASCIIChar) which represents a single ASCII character as an object

```
#include <iostream.h>     // for cin and cout
class ASCIIChar
{
  protected:
    char character;
  public:
    void getChar(int char_number);
    void showChar();
};
void ASCIIChar::getChar(int char_number)
{
// the ASCII codes for upper and lower case letters
  if((char_number >= 65 && char_number <= 90) ||
     (char_number >= 97 && char_number <= 122))
     {  character = char_number;   }
  else
     {  character = 32;   }
}
void ASCIIChar::showChar()
{// if 'char number' is not in a letter range, a space will
// be stored
cout < < "Character is " < < character < < endl;
}
```

A Simple Program

```
#include "ascichar.h"
void main()
{
  ASCIIChar a_char;
  a_char.getChar(97);
// 97 is the ASCII code for 'a'
  a_char.showChar();
}
```

- Using the ASCIIChar class as a base class we are now going to define a derived class called UpperCaseChar.
- As the base class has a getChar method which eliminates any characters which are not required it would be wasteful to re-write the getChar method in the derived class
- Instead we can extend the base getChar method to add extra functionality

Extension of a base method

- Instead we can extend the base `getChar` method to add extra functionality
- This is done using the scope resolution operator (**::**) in the following way
  `classname::method(parameter list)`
- The **`classname`** is required otherwise the compiler will call the method declared in the derived class which would cause an infinite loop due to recursion

```
#include "ascichar.h"
class UpperCaseChar : public ASCIIChar
{
public:
     void getChar(int char_in);
};
void UpperCaseChar::getChar(int char_in)
{
// the base class method is called
ASCIIChar::getChar(char_in);

// if 'character' is not a space, it is a letter
if(character != 32)
   {
   // 97 to 122 are lower case letters
   if(character >= 97)
      {
      // the upper and lower case ranges are
      //32 apart in the table
      character -= 32;
      }
   }
}
```

Run-time Polymorphism

Oil Pump

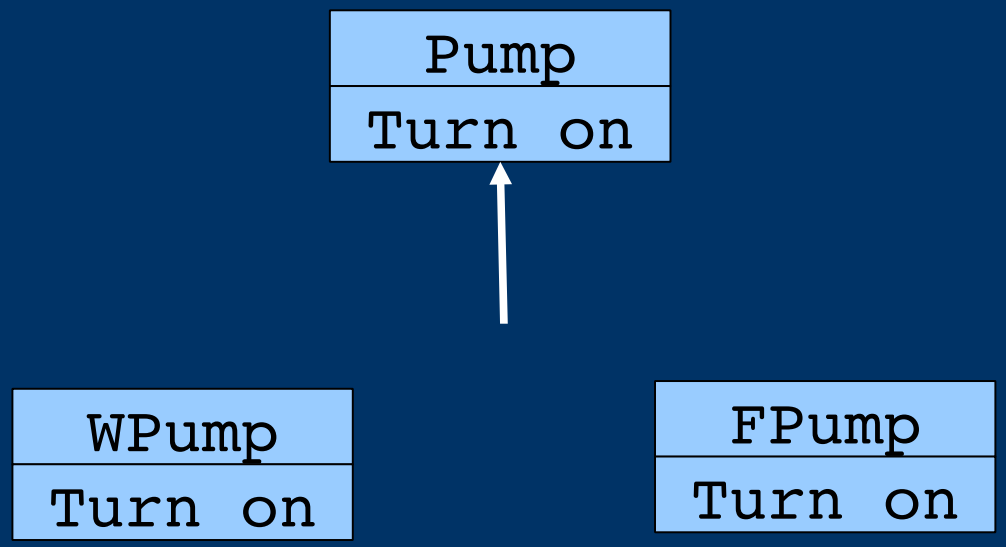
easily counted & named
able to predict their persistence

Rain drop

Only identified at rain-time

Some objects are identified at compile time whereas others can only be identified at run time

Dynamic Binding



Run-time Polymorphism

- When we know the class of an object we know how it will respond to a particular message
- In this case the compiler binds in the correct method for the object at compile time
- But what happens if we wish to create a class where we do not know what object will be instantiated until run time?
- In C++ this is possible by the use of base class pointers (static identifiers) and the dynamic binding of methods.

The role of the abstract base class (1)

- The base class in a classification hierarchy represent the common characteristics of the derived objects
- However this is not specialised enough to represent an object in it's own right
- We mentioned previously that we could define abstract methods in the base class which could be used as place holders for the method names in the derived classes

The role of the abstract base class (2)

- Now when we wish to use dynamic objects we create a pointer to the base class and then decide which derived class we wish to use by instantiating the object using the **`new`** operator.
- As the base class has an abstract method of the correct type the compiler will compile without error.
- However we still need some way of telling the program which version of the method to use in the program.

C++ syntax

- In C++ we use the keyword '**`virtual`**' to denote those methods which may be dynamically bound at run-time
- For the following example we will use the following class hierarchy

```
#include <iostream.h>
class FlyingMachine
{
   public:
      void showName();
};
class Helicopter : public FlyingMachine
{
   public:
      void showName();
};
class Aeroplane : public FlyingMachine
{
   public:
      void showName();
};
void FlyingMachine::showName()
{    cout < < "Flying Machine" < < endl; }
void Helicopter::showName()
{    cout < < "Helicopter" < < endl; }
void Aeroplane::showName()
{    cout < < "Aeroplane" < < endl; }
```

Example Program

```
#include "flying.h"
void main()
{
  FlyingMachine* flyer;
  flyer = new Helicopter;
  flyer -> showName();
  delete flyer;
  flyer = new Aeroplane;
  flyer -> showName();
  delete flyer;
}
```

- Which gives the following output :

  `Flying Machine`
  `Flying Machine`

- What is wrong with this output ?

Problems

- The compiler has statically bound the methods from the base class and not the methods in the derived class.
- This means that we only call the base class methods which prints Flying Machine instead of the correct name for the object.
- By default the compiler will always use this static binding when building programs so we have to use the **`virtual`** keyword to tell the compiler not to use the method until run-time when it can decide which method to use in the correct circumstance.

Solution

- We now re-write the class method definitions as follows

```
class FlyingMachine
{
   public :
      virtual void showName();
};
class Helicopter : public FlyingMachine
{
   public :
      virtual void showName();
};
class Aeroplane : public FlyingMachine
{
   public :
      virtual void showName();
};
```

The virtual Destructor

- When we call the delete method on an object the destructor is called
- However when using dynamic object only the destructor from the base class will be called as it will be statically bound at compile time.
- To overcome this problem we may also create a virtual destructor to destroy any of the objects attributes and memory allocation.
- This is declared as follows:

```
virtual ~FlyingMachine();

virtual ~Helicoptor();

virtual ~Aeroplane();
```

Pure Virtual Methods

- At present we have declared our methods in base class so that they do not contain any code but may be used.
- This has a limitation in the fact that if the derived object does not have an overriding method the base class method is used.
- To force the compiler to ensure that all derived classes must have a method of the same name we must declare the base class as a pure **`virtual`** method.

Pure Virtual Methods (2)

- This is done using the following C++ syntax

```
virtual void showName() = 0;
```

- If we now modify the Aeroplane class to remove the showName method we get the following error message (from gnu C++)

```
flying2.cpp:12: error: cannot allocate an object of
abstract type 'Aeroplane'
flyingmachine2.h:16: note:   because the following
virtual functions are pure within 'Aeroplane':
flyingmachine2.h:21: note:  virtual void
FlyingMachine::showName()
```