Virtual Function and Polymorphism allow the design and implementation of systems that are easily extensible…

Programs written to generically process objects of all existing classes in a hierarchy…

This can be achieved by taking the advantage of the "***pointer to derived class***" compatibility with a "***pointer to its base class***"…

<Pointer-to-base-class> = <Pointer-to-derived-class>
<Pointer-to-base-class> = & <Object-of-derived-class>

As seen earlier, a ***pointer*** to the base-class can point to any object of its derived classes without the need of explicit casting…

The same rule applies for a base-class ***reference*** to a derived object…

Lets consider the following scenario…

```
#include <iostream.h>
class pet {
 public:
   void speak() {cout << "can't speak \n";}
};
class dog : public pet {
 public:
   void speak() { cout << "wooffff! \n"; }
};
class cat : public pet {
 public:
   void speak() { cout << "meawwwww! \n"; }
};
class bird : public pet {
 public:
   void speak() { cout << "squeekkk! \n"; }
};
```

```
void main() {
   dog spike;
   cat tom;
   bird tweety;
   pet *cage= &tom ;
   spike.speak();
   tweety.speak();
   tom.speak();
   cage->speak();[i1]
   static_cast<bird*>(cage)->speak();[i2]
}
```


What about this scenario ???…

```
void main() {
   pet *cage[3];
   cage[0] = new dog;
   cage[1] = new cat;
   cage[2] = new bird;
   cage[1]->speak();[i3]
   static_cast<bird*>(cage[1])->speak();[i4]
}
```


A solution !!!…

Virtual Functions

```
#include <iostream.h>
class pet {
 public:
   virtual[IAN5] void speak() {cout << "can't speak
\n";}
};
class dog : public pet {
 public:
   void speak() { cout << "wooffff! \n"; }
};
class cat : public pet {
 public:
   void speak() { cout << "meawwwww! \n"; }
};
class bird : public pet {
 public:
   void speak() { cout << "squeekkk! \n"; }
};
```

```
void sayIt(pet &pRf, pet *pPr) {
    pRf.speak();            // Dynamic binding.[IAN6]
    pPr->speak();           // Dynamic binding.[IAN7]
}
```

```
void main() {
   bird tweety;
   pet *cage[3], &pRef_1 = tweety;

   cage[0] = new dog;
   cage[1] = new cat;
   cage[2] = new bird;

   cage[2]->speak();        // Dynamic binding.[JAN8]
   cage[0]->speak();
   cage[1]->speak();

   pRef_1.speak();          // Dynamic binding.[JAN9]

   tweety.speak();          // Static binding.[JAN10]

   pet &pRef_2 = *cage[1];
   pRef_2.speak();          // Dynamic binding.[JAN11]

   sayIt(*cage[0], cage[2]);
}
```

- By declaring function ***speak*** as ***virtual***, and by using:
  - a base-class pointer that points to a derived-class object…
  - a base-class reference that references a derived class object…

  the program will chose the correct derived class ***speak*** dynamically at run-time… ***Dynamic binding***…

- When a virtual function is called by the object name using the "dot-operator"… the function called is the one ***defined for*** (or ***inherited by***) the class of that particular object… ***Static binding***…

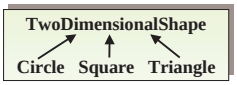
More realistic scenario…

```
#include <iostream.h>
class Polygon {
 public:
   void setPoly (int a, int b) {
      width = a;
      height = b;
   }
   virtual[JAN12] int area() {return 0;}
 protected:
   int width, height;
};

class Rectangle: public Polygon {
 public:
   int area () {return (width * height);}
};

class Triangle: public Polygon {
 public:
   int area() {return  (width * height / 2);}
};
```

```
void main () {
   Rectangle rect;
   Polygon *polyPtr[2], &polyRef = rect ;

   polyPtr[0] = new Rectangle;
   polyPtr[1] = new Triangle;

   polyPtr[0]->setPoly(4,5);
   polyPtr[1]->setPoly(2,50);
   polyRef.setPoly(1,30);

   cout << polyPtr[0]->area()[JAN13] << endl;
   cout << polyPtr[1]->area() << endl;
   cout << polyRef.area()[JAN14] << endl;
}
```


Abstract and Concrete Classes

- ***Abstract classes***:
  - Sole purpose is to provide a base class for other classes… Too generic to define real objects…



  - ***No objects*** of an abstract base class can be instantiated… Syntax error…
  - ***Can have*** pointers and references…

- ***Concrete classes***:
  - Classes that can instantiate objects…
  - Provide specifics to make real objects… Square, Circle, etc…

- A class is made ***abstract*** by declaring one or more of its virtual function to be "***pure***"…
  A ***pure virtual*** function is one with an ***initializer of =0*** in its declarations…

```
class Polygon {
 public:
   void setPoly (int a, int b) {
      width = a;
      height = b;
   }
   virtual int area()  =0;
   virtual void print() const;[JAN15]
 protected:
   int width, height;
};
```



```
#include <iostream.h>
class Polygon {
 public:
   void setPoly (int a, int b) {
      width = a;
      height = b;
   }
   virtual int area() =0;
   virtual void print()=0;
 protected:
   int width, height;
};

class Rectangle: public Polygon {
 public:
   int area () {return (width * height);}
   void print() {
      cout << width << endl;
      cout << height << endl;
   }
};

class Triangle : public Polygon[JAN16] {
 public:
   int area() {return  (width * height / 2);}
};


void main () {
   Polygon poly;[JAN17]
   Rectangle rect;
   Triangle trgl;[JAN18]
}
```

Syntax Error…

Polymorphism

- From the Greek:
  - **Poly** → **Many**
  - **Morph** → **Shape**
- In OOP:
  - Ability for objects of different classes (related by inheritance) to respond differently to the same function call…
- ***Polymorphism*** is implemented via **virtual** functions…
  - When a request is made trough a ***base class pointer*** (or ***reference***) to use a virtual function, C++ chooses the correct overridden function in the appropriate derived class associated with the object…
- Needless to say that **non-virtual** function behave in the same manner discussed in the inheritance chapter…
  - If ***non-virtual*** member function defined in multiple classes and called from **base-class** pointer then the **base-class** version is used…
  - If called from **derived-class** pointer then **derived-class** version is used…
  - ***Non-polymorphic*** behavior…

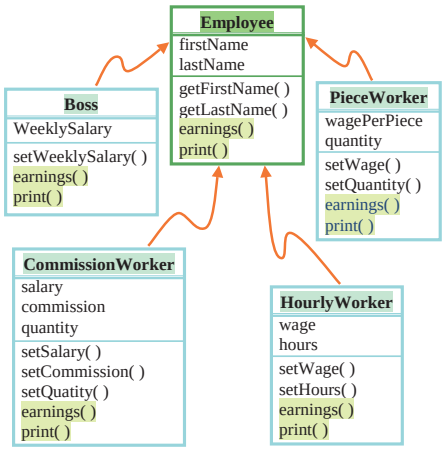

```
#include <iostream.h>

class Polygon {
 public:
   void setPoly (int a, int b) {
      width = a;
      height = b;
   }
   virtual int area() { return 0;}
   void print() const {
      cout << "Width:" << width << " - "
           << "Height:" << height << endl;
   }
 protected:
   int width, height;
};

class Rectangle: public Polygon {
 public:
   int area() {return (width * height);}
};

class Triangle: public Polygon {
 public:
   int area() {return (width * height / 2);}
   void print() const {
      cout << "Base:" << width << " - "
           << "Height:" << height << endl;
   }
};
```



```
void main() {
   Rectangle rect;
   Triangle trgl;
   Polygon *polyPtr,
      &polyRefR = rect,
      &polyRefT = trgl;

   rect.setPoly(10,2);
   trgl.setPoly(30,3);

   rect.print();
   trgl.print();
   cout << rect.area() << endl;
   cout << trgl.area() << endl;

   polyPtr = &trgl;
   polyPtr->print();
   cout << polyPtr->area() << endl;

   polyPtr = &rect;
   polyPtr->print();
   cout << polyPtr->area() << endl;

   polyRefT.print();
   cout << polyRefT.area() << endl;

   polyRefR.print();
   cout << polyRefR.area() << endl;
}
```


Case Study: A Payroll system using Polymorphism

- Perform different payroll calculations for different types of employees (base class ***Employee***):
  - ***Boss*** - fixed weekly salary
  - ***CommissionWorker*** - flat base salary + commission
  - ***PieceWorker*** - paid according to items produced
  - ***HourlyWorker*** - hourly wage + overtime



Virtual destructor

- Recall

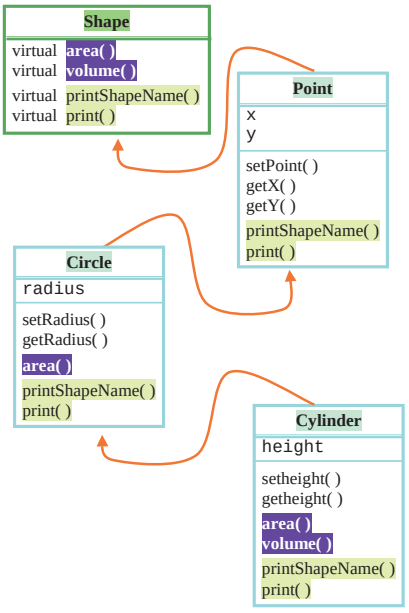
```
#include <iostream.h>
class A {
 public:
   A(int x = 0) {
   valA = x;
   cout << "Constructor: valA= "
        << valA <<endl;
   }
   ~A() {
      cout << "Destructor: valA= "
         << valA << endl;
   }
 private:
   int valA;
};

void main() {
   A *aPtr, aObj(5);
   aPtr = new A[2];
   delete [] aPtr;
}[JAN20]
```

- An object allocated using the ***new*** operator must be explicitly deallocated using the ***delete*** operator… It is at the ***deletion*** time that the ***destructors*** is called…


```
#include <iostream.h>
class basA {
 public:
   basA (int a) { x = a; }
   virtual[JAN21] ~basA() {
      cout << "Base destructor... x = " << x << endl;
   }
 protected:
   int x;
};
class drvA: public basA {
 public:
   drvA(int a, int b) : basA(a) {y = b;}
   int getA() {return y;}
   void setA(int a) { y = a;}
   ~drvA() {
      cout << "Derived destructor... y = " << y <<endl;
   }
 private:
   int y;
};

void main() {
   basA *basPtr;
   drvA *drvPtr;
   drvPtr=new drvA(10, 20);
   delete drvPtr;
   drvPtr=new drvA(100, 200);
   basPtr = drvPtr;
   delete basPtr;[JAN22]
}
```


- Problem:
  - If a ***base-class pointer*** to a ***derived object*** is **deleted**, the ***base-class destructor*** will act on the object…
- Solution:
  - Declare a ***virtual base-class destructor*** to ensure that the appropriate destructor will be called

Case Study: Inheriting Interface and Implementation

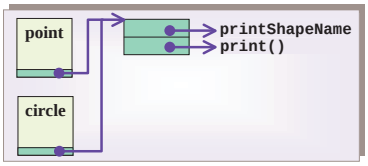


Polymorphism, virtual Functions and Dynamic Binding "Under the Hood"

The fundamental idea behind polymorphism is that the compiler **does not know** which function to call **compile-time**…the appropriate function will be selected ***run-time***….

A common implementation is the following…

- An ***object*** containing virtual member functions holds as its ***first data member*** a ***hidden field***, pointing to an ***array of pointers*** containing the ***addresses of the virtual member functions*** …



- The hidden data member is usually called the ***vpointer***…
- The array of virtual member function addresses is called the ***vtable***…

- Final Note:
  - Polymorphism requires some overhead…
  - Polymorphism is not used in STL (Standard Template Library) to optimize performance…