Lezione 5

Ereditarietà
Polimorfismo

Ereditarietà

- L'ereditarietà è importante per la creazione di software riutilizzabile e per controllare la complessita' del codice
- classi nuove sono progettate sulla base di classi pre-esistenti
- le nuove classi acquisiscono gli attributi e i comportamenti delle classi precedenti ed aggiungono caratteristiche nuove o raffinano caratteristiche pre-esistenti

Il Polimorfismo

- Il polimorfismo permette di scrivere programmi in modo generale, astratto
- si riescono a trattare una ampia gamma di classi correlate
- si possono trattare anche classi non ancora specificate

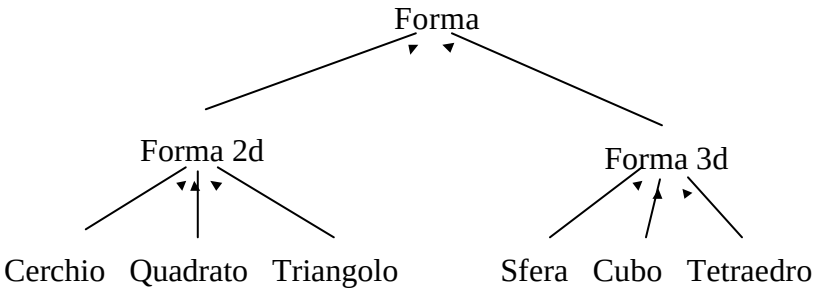
Ereditarietà

- Quando si crea una nuova classe si può fare in modo che questa erediti i dati membro e le funzioni membro da una classe già definita precedentemente
- la classe precedente prende il nome di *classe base*
- la classe che eredita prende il nome di *classe derivata*
- è possibile continuare il procedimento creando una classe che eredita a sua volta da una classe derivata
- questo procedimento crea una gerarchia
- una classe base può essere *diretta* o *indiretta* se si trova a livelli più alti della gerarchia di derivazione

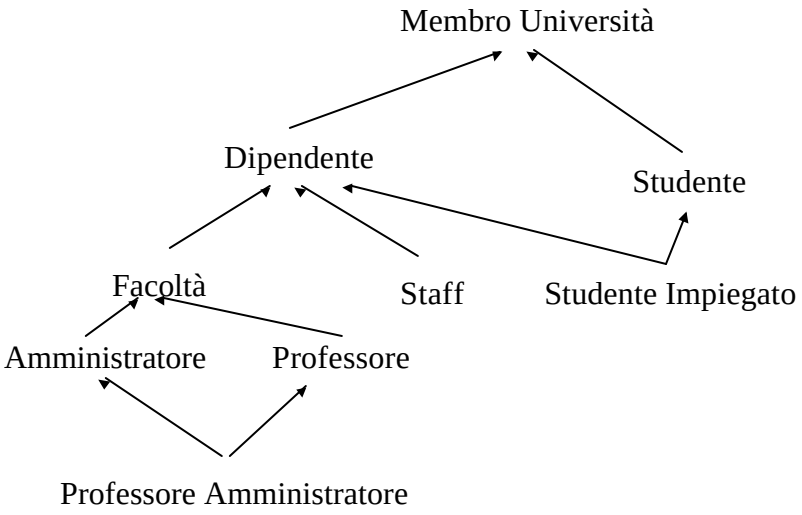
Ereditarietà

- Esistono due tipi di ereditarietà:
  - C *singola*: quando una classe derivata eredita da una sola classe base
  - – *multipla*: quando una classe derivata eredita da più classi base che tra loro possono non essere correlate

Ereditarietà singola



Ereditarietà multipla



Cosa può ereditare una classe?

- la classe derivata acquisisce i
  dati membro
- e le
  funzioni membro
- della classe base

La sintassi

- Nella definizione della classe derivata si aggiunge la specifica della classe base da cui si eredita ed il tipo di eredità

```
class BaseClass{
  //dichiarazione
};

class DerivedClass: public BaseClass{
  //dichiarazione
};
```

- Esistono tre tipi di ereditarietà:
  public, private, protected

Costruttore di classe derivata

- Il costruttore della classe derivata si deve occupare di inizializzare i dati membri aggiuntivi
- si può utilizzare il costruttore della classe base per inizializzare i dai membri condivisi con la classe base
- la sintassi e:

```
NomeClassDeriv(T prm_bas,T prm_drv):NomeClassBase(prm_bas){
    //init con prm_drv
}
```

Dichiarazione di una classe base

```
#ifndef POINT_H
#define POINT_H

class Point{
   friend ostream operator<<(ostream &, const Point &);

public:
   Point(int=0, int=0);
   void setPoint(int, int);
   int getX() const {return x;}
   int getY() const {return y;}

protected:
   int x,y;
};

#endif
```

Definizione delle funzioni

```
#include<iostream>
#include “point.h”

Point::Point(int a, int b){set(a,b);}

void Point::set(int a, int b){x=a;y=b;}

ostream& operator<<(ostream &out, const Point &p){
   out<<“[”<<p.x<<“,”<<p.y<<“]”<<endl;
   return out;
}
```

Dichiarazione di una classe derivata

```
#ifndef CIRCLE_H
#define CIRCLE_H

class Circle: public Point{
   friend ostream& operator<<(ostream &, Circle &);
public:
   Circle(double r=0.0, int x=0, int y=0);
   void setRadius(double);
   double getRadius() const {return radius;}
   double area() const;
protected:
   double radius;
}

#endif
```

Definizione delle funzioni

```
#include<iostream>
#include "circle.h"
 Circle::Circle(double r, int a, int b): Point(a,b){//Costruttore
   per la classe base
   setRadius(r);
}
void Circle::setRadius(double r){
   radius=(r>=0 ? r: 0);
}
double Circle::area()const{
   return 3.14159 * radius *radius;
}
ostream & opertor<<(ostream &out, Circle &c){
   out<<"Center:"<< static_cast<Point>( c )<<"//cast esplicito
   Radius:"<<c.radius<<endl;
   return out;
}
```

Esempio

```
#include<iostream>
#include “point.h”
#include “circle.h”
int main(){
   Point p(10,20);
   Circle c(2.1,30,40);
   cout<<c; //Stampa: Center: [30,40] Radius:2.1
   Point *pPtr=&c;
   cout<<(*pPtr); // Stampa: [30,40] il punt vede solo la i
   dati membri della classe base
   Circle *cPtr=static_cast<Circle *>(pPtr);
   cout<<(*cPtr); // Stampa: Center: [30,40] Radius:2.1. I
   dati ci sono sempre. Erano solo non visibili prima
   cPtr=static_cast<Circle *>(&p);
   cout<<(*cPtr); // Stampa: Center: [30,40] Radius:??? Adesso
   invece non sono mai esistiti e quindi si va ad accedere in
   memoria casualmente
}
```

Specializzazione

- la classe derivata aggiunge dati membro e funzioni membro a quelle della classe base
- la classe derivata specializza, raffina, reimplementa le funzioni membro della classe base
- una classe derivata è più *grande* di una classe base nel senso che occupa più spazio, ha più dati e funzioni membro
- una classe derivata rappresenta tuttavia un gruppo più *ristretto* di oggetti, è più specializzata

Overriding di funzioni membro

- Una classe derivata può ridefinire una funzione membro della classe base
- Attenzione non è overloading: infatti la funzione ha lo stesso nome e gli stessi parametri
- se fosse stato un caso di overloading la nuova funzione si sarebbe dovuta distinguere per qualche parametro
- la redefinizione si chiama *overriding*
- la funzione nella classe base è mascherata dalla funzione ridefinita nella classe derivata

Esempio

```
Class Base{
public:
   Base(int a=0){dato=a;}
   void print(){cout<<dato;}
private:
   int indice;
};

class Derivata: public Base{
public:
   Derivata(double b=0.0, int a=0):Base(a){num=b;}
   void print(){cout<<num;}
private:
   double num;
};
int main(){
   Derivata obj(2.3,88);
   obj.print(); //Stampa: 2.3 e non 88
   return 0;
}
```

Overriding di funzioni membro

- Una classe derivata può aver bisogno di accedere alle funzioni della classe base
- se le funzioni sono state ridefinite tramite overriding sorge il problema di indicarle senza ambiguità
- lo si può fare utilizzando l'operatore di risoluzione :: ed indicando il nome della classe base

Esempio

```
Class Base{
public:
   Base(int a=0){dato=a;}
   void print(){cout<<dato;}
protected:
   int indice;
};
class Derivata: public Base{
public:
   Derivata(double b=0.0, int a=0):Base(a){num=b;}
   void print(){Base::print(); cout<<" "<<num;}
//Nota: indice è accessibile, allora va bene: cout<<indice<<" "<<num;
private:
   double num;
};
int main(){
   Derivata obj(2.3,88);
   obj.print(); //Stampa: 88 2.3
   return 0;
}
```

Protezione di ereditarietà

- L'eredità in C++ può essere di tre tipi:
  - pubblica (**`public`**)
  - privata (**`private`**)
  - protetta (**`protected`**)
- si distinguono in base alle restrizioni di accesso che si realizzano su i dati membro e le funzioni membro ereditate

Significato intuitivo

- Ereditare in modo pubblico significa che ciò che prima era visibile all'esterno rimane visibile all'esterno anche dopo l'eredità
- Ereditare in modo protetto o privato significa che ciò che prima era visibile all'esterno rimane visibile solo all'interno della classe che eredita

Eredità pubblica:

- ciò che è pubblico nella classe base *è* accessibile alla classe derivata ed *è* accessibile all'esterno
- ciò che è protetto nella classe base *è* accessibile alla classe derivata ma *non è* accessibile all'esterno
- ciò che è privato nella classe base *non è* accessibile alla classe derivata e *non è* accessibile all'esterno

Eredità pubblica:

```
Class Base{
public:
   Base(int usr_a=0, int usr_b=0, int usr_c=0)
   {a_pub=usr_a;b_pro=usr_b;c_pri=usr_c;}
   int a_pub;
protected:
   int b_pro;
private:
   int c_pri;
};
```

Eredità pubblica:

```
class Derivata: public Base{
public:
   Derivata(int usr_a=0, int usr_b=0, int usr_c=0, int
   usr_a_d=0, int usr_c_d=0)
   :Base(usr_a,usr_b,usr_c)
   {a_d=usr_a_d;c_d=usr_c_d;}
   int get_a(){return a_pub;}
   int get_b(){return b_pro;}
   int get_c(){return c_pri;}
   int a_d_pub;
private:
   int c_d_pub;
};
```

Eredità pubblica:

```
Void main(){
   Derivata obj(1,2,3,4,5);

   obj.a_pub=10;//OK
   obj.b_pro=10;//NO
   obj.c_pri=10;//NO
   obj.a_d_pub=10;//OK
   obj.c_d_pri=10;

   obj.get_a();//OK
   obj.get_b();//OK
   obj.get_c();//NO la classe derivata non ha accesso al dato
   privato della classe base
}
```

Eredità protetta:

- ciò che è pubblico nella classe base *è* accessibile alla classe derivata ma *non è* accessibile all'esterno
  (ma può essere tramandato ai discendenti in modo che questi possano accedervi)
- ciò che è protetto nella classe base *è* accessibile alla classe derivata ma *non è* accessibile all'esterno

- ciò che è privato nella classe base *non è* accessibile alla classe derivata e *non è* accessibile all'esterno

Eredità protetta:

```
Void main(){
   Derivata obj(1,2,3,4,5);

   obj.a=10;//NO
   obj.b=10;//NO
   obj.c=10;//NO
   obj.a_d=10;//OK
   obj.c_d=10;//NO

   obj.get_a();//OK
   obj.get_b();//OK
   obj.get_c();//NO
}
```

Eredità privata:

- ciò che è pubblico nella classe base *è* accessibile alla classe derivata e *non è* accessibile all'esterno
  (né è accessibile ai discendenti)
- ciò che è protetto nella classe base *è* accessibile alla classe derivata ma *non è* accessibile all'esterno

- ciò che è privato nella classe base *non è* accessibile alla classe derivata e *non è* accessibile all'esterno

Eredità privata:

```
Void main(){
   Derivata obj(1,2,3,4,5);

   obj.a=10;//NO
   obj.b=10;//NO
   obj.c=10;//NO
   obj.a_d=10;//OK
   obj.c_d=10;//NO

   obj.get_a();//OK
   obj.get_b();//OK
   obj.get_c();//NO
}
```

Eredità protetta e privata:

- i membri pubblici e protetti della classe base se ereditati in modo protetto diventano protetti nella classe derivata
- i membri pubblici e protetti della classe base se ereditati in modo privato diventano privati nella classe derivata

Eredità protetta e privata:

- La differenza fra l'ereditarietà protetta o privata è rilevabile solo se la classe derivata è a sua volta ereditata da una o più classi gerarchicamente
- nel caso di eredità protetta i membri pubblici e protetti rimangono accessibili per tutta la gerarchia (ma non all'esterno)
- mentre nel caso di eredità privata i membri di qualsiasi tipo cessano di essere trasmessi in modo accessibile ai discendenti

Eredità protetta e privata:

- Esempio di utilità del meccanismo di ereditarietà privata
  - data una classe ListClass che manipola le liste si può derivare una classe QueueClass che la eredita in modalità privata
  - tutti i metodi di ListClass diventano privati di QueueClass e non accessibili all'esterno
  - si definiscono i metodi pubblici di QueueClass che non fanno altro che richiamare i metodi di ListClass di interesse
    - ex: enqueue chiama insertAtBack e dequeue chiama removeFromFront
  - gli altri metodi rimangono inaccessibili

Costruttori e distruttori in classi derivate

- Dato che una classe derivata contiene i membri della classe base quando viene istanziata deve poter accedere al costruttore della classe base
- Sintassi:

```
ClasseDerivata(arg_base, arg_deriv):ClasseBase(arg_base){...}
```

- Se non viene fatto in modo esplicito allora la classe derivata chiama in modo implicito il costruttore di default della classe base

Costruttori e distruttori in classi derivate

- Dato che il distruttore viene invocato automaticamente e non prende parametri, la classe derivata non ha un modo esplicito per invocare il distruttore della classe base

Ordine di chiamata dei Costr/Distr

```
#include<iostream>
class Base{
public:
   Base(){cout<<"Costruzione Base\n";}

   ~Base(){cout<<"Distruzione Base\n";}
};
class Deriv1:public Base{
public:
   Deriv1(){cout<<"Costruzione
   Deriv1\n";}
   ~Deriv1(){cout<<"Distruzione
   Deriv1\n";}
};
class Deriv2:public Deriv1{
public:
   Deriv2(){cout<<"Costruzione
   Deriv2\n";}
   ~Deriv2(){cout<<"Distruzione
   Deriv2\n";}
};
```

```
main(){Deriv2 ob;}
```

```
Output:

Costruzione Base
Costruzione Deriv1
Costruzione Deriv2
Distruzione Deriv2
Distruzione Deriv1
Distruzione Base
```

Nota sui costruttori

- Si ha un oggetto di una classe derivata D da una classe base B. Sia B che D contengono oggetti di altre classi CB e CD rispettivamente
- quando si crea un oggetto di tipo D sono eseguiti prima i costruttori degli oggetti CB poi il costruttore di B, poi i costruttori degli oggetti CD e infine il costruttore di D
- I distruttori sono chiamati in ordine inverso rispetto ai corrispondenti costruttori

Eredità singola o multipla

- ereditarietà singola: la classe derivata eredita solo da una classe base
- ereditarietà multipla: la classe derivata eredita da più classi base (anche fra di loro eterogenee)

Eredità multipla

```
class Base1{
public:
   Base1(int i=0){b1=i;}
private:
   int b1;
};

class Base2{
public:
   Base2(int i=0){b2=i;}
private:
   int b2;
};

class Deriv:public Base1, public Base2{
public:
   Deriv(int i=0, int j=0, int z=0):Base1(i), Base2(j){d=z;}
private:
   int d;
};
```

Relazioni fra classi

- Due classi possono essere in relazione l'una con l'altra nei modi:
  - **è un**
  - **ha un**

Relazione: è un

- definizione: una classe è una specializzazione di una seconda classe
- implementazione: tramite il meccanismo di ereditarietà
- Es: una classe cerchio è una classe punto (a cui si è aggiunto dati e membri)

Relazione: ha un

- definizione: una classe ha nella sua composizione altre classi
- implementazione: una classe ha fra i suoi dati membro altre classi
- Es: una classe Impiegato ha fra i propri attributi la classe Anagrafica e la classe Azienda

Il polimorfismo

Introduzione al polimorfismo

- Il polimorfismo è la possibilità di utilizzare una unica interfaccia per più metodi

Polimorfismo

- il polimorfismo al momento della compilazione si ottiene con con l'overloading
- il polimorfismo al momento dell'esecuzione si ottiene con l'ereditarietà e le funzioni virtuali

Esempio

- classe base Forma
- classe derivata Punto
- classe derivata Cerchio
- classe derivata Cilindro

- si vuole poter invocare una unica funzione **`disegna`** per tutte le classi derivate
- in questo modo un vettore che contiene puntatori a oggetti di diverse classi può chiamare lo stesso metodo **`(*ptr).disegna`** su ogni elemento

Utilità

- Un modo alternativo per selezionare una azione dato un oggetto di tipo diverso è utilizzare la commutazione (switch)
- svantaggi:
  - si deve prevedere esplicitamente il test per ogni tipo
  - si devono testare tutti i casi possibili
  - quando si aggiunge un tipo nuovo vanno modificati gli switch in tutto il programma
- la programmazione polimorfica elimina questi svantaggi e dà al programma un aspetto semplificato (meno diramazioni e più codice sequenziale)

Utilità

- Si crea una gerarchia di classi
- dal caso più generale a quello più specifico
- tutte le funzionalità e i metodi comuni di interfacciamento sono definiti all'interno di una classe base
- quando i metodi possono essere implementati solo dalle classi derivate si utilizzano le funzioni virtuali per specificare l'interfaccia che deve essere garantita

Funzioni virtuali

- I metodi di una classe base possono essere dichiarati **`virtual`**

  ```
  virtual retType funcName(parType);
  ```

- una classe che eredita una funzione virtual può ridefinirla
- ereditare una funzione virtual è diverso dal *overriding* di funzione
- la differenza si nota quando si utilizzano i puntatori alla classe base per riferirsi agli oggetti delle classi derivate

Esempio non-virtual

```
class Base{
public:
   Base(int i=0){b=i;}
   void print(){cout<<“Base:”<<b;}
private:
   int b;
};

class Deriv:public Base{
public:
   Deriv(int i=0, int
   z=0):Base(i){d=z;}
   void print(){cout<<“Deriv:”<<d;}
private:
   int d;
};
```

```
void main(){
   Deriv dObj(10,20);
   Base* objPtr=&dObj;

   objPtr->print();
   //Stampa: Base:10
}
```

Esempio virtual

```
class Base{
public:
   Base(int i=0){b=i;}
   virtual void print(){cout<<“Base:”<<b;}
private:
   int b;
};
class Deriv:public Base{
public:
   Deriv(int i=0, int z=0):Base(i){d=z;}
   void print(){cout<<“Deriv:”<<d;}
private:
   int d;
};
void main(){
   Deriv dObj(10,20);
   Base* objPtr=&dObj;
   objPtr->print(); //Stampa: Deriv:20
}
```

Funzioni virtuali

- In pratica accedendo ad un oggetto di una classe derivata tramite un puntatore di tipo classe base
  - nel caso ordinario si accede ai membri della classe base
  - nel caso virtual si accede ai membri della classe derivata

Funzioni virtuali pure

- Talvolta non è possibile definire un comportamento significativo per una funzione in una classe base
- Ex: la classe base Forma può avere un metodo Stampa ma questo è definibile con precisione solo dalle classi derivate Cerchio e Cilindro che specificano i propri attributi

Funzioni virtuali pure

- Se si vuole specificare che le classi che erditano devono *necessariamente* definire una funzione allora si rende tale funzione una

  funzione virtuale pura

- Sintassi

```
virtual retType FuncName(argType)=0;
```

Classi astratte

- Una classe che contiene una o più funzioni astratte pure è una *classe astratta*
- Non è possibile istanziare alcun oggetto di una classe astratta
- infatti esiste almeno un metodo che non è definito!!

Classi astratte

- Una gerarchia non deve necessariamente contenere classi astratte
- Tuttavia il livello più alto (o i primi livelli) generalmente è realizzato come classe astratta
- Si specifica così l'interfaccia necessaria per tutte le classi derivate

Distruttori virtuali

- Caso: si allocano dinamicamente oggetti e si deallocano tramite l'operatore **`delete`**
- si sta usando un puntatore alla classe base per riferirsi ad un oggetto di una classe derivata
- allora viene chiamato il distruttore della classe base e non quello della classe derivata

- per chiamare correttamente il distruttore della classe derivata si deve dichiarare il distruttore come **`virtual`**

Distruttori virtuali: note

- Per classi con funzioni virtuali si consiglia di creare sempre distruttori virtuali anche se non strettamente necessari.
- In questo modo le classi derivate invocheranno i distruttori in modo appropriato

- i costruttori non possono essere virtuali

Esempio

```
// Definition of abstract base class Shape
#ifndef SHAPE_H
#define SHAPE_H

class Shape {
public:
   virtual double area() const { return 0.0; }
   virtual double volume() const { return 0.0; }

   // pure virtual functions overridden in derived classes
   virtual void printShapeName() const = 0;
   virtual void print() const = 0;
};

#endif
```

```
// Definition of class Point
#ifndef POINT1_H
#define POINT1_H
#include <iostream>
#include "shape.h"

class Point : public Shape {
public:
   Point( int = 0, int = 0 );
void setPoint( int, int );
   int getX() const { return x; }
   int getY() const { return y; }
   virtual void printShapeName() const
        { cout << "Point: "; }
   virtual void print() const;
private:
   int x, y;
};

#endif
```

```
// Member function definitions for class Point
#include "point1.h"

Point::Point( int a, int b ) { setPoint( a, b ); }

void Point::setPoint( int a, int b )
{
   x = a;
   y = b;
}

void Point::print() const
   { cout << '[' << x << ", " << y << ']'; }
```

```
// Definition of class Circle
#ifndef CIRCLE1_H
#define CIRCLE1_H
#include "point1.h"

class Circle : public Point {
public:
   // default constructor
   Circle( double r = 0.0,
        int x = 0, int y = 0 );

   void setRadius( double );
   double getRadius() const;
   virtual double area() const;
   virtual void printShapeName() const
        { cout << "Circle: "; }
   virtual void print() const;
private:
   double radius;   // radius of Circle
};

#endif
```

```
// Member function definitions for class Circle
#include <iostream>
using std::cout;
#include "circle1.h”

Circle::Circle( double r, int a, int b )
   : Point( a, b )
{ setRadius( r ); }
void Circle::setRadius( double r )
   { radius = r > 0 ? r : 0; }
double Circle::getRadius() const
   { return radius; }
double Circle::area() const
   { return 3.14159 * radius * radius; }
void Circle::print() const{
   Point::print();
   cout << "; Radius = " << radius;
}
```

```
// Definition of class Cylinder
#ifndef CYLINDR1_H
#define CYLINDR1_H
#include "circle1.h"

class Cylinder : public Circle {
public:
   // default constructor
   Cylinder( double h = 0.0,
double r = 0.0,int x = 0, int y = 0 );

   void setHeight( double );
   double getHeight();
   virtual double area() const;
   virtual double volume() const;
   virtual void printShapeName() const {cout <<
"Cylinder: ";}
   virtual void print() const;
private:
   double height;};

#endif
```

```
// Member and friend function definitions for class Cylinder
#include <iostream>
using std::cout;
#include "cylindr1.h"
Cylinder::Cylinder( double h, double r, int x, int y )
: Circle( r, x, y )
{ setHeight( h ); }
void Cylinder::setHeight( double h )
   { height = h > 0 ? h : 0; }
double Cylinder::getHeight() { return height; }
double Cylinder::area() const{
   return 2 * Circle::area() +
          2 * 3.14159 * getRadius() * height;}
double Cylinder::volume() const
   { return Circle::area() * height; }
void Cylinder::print() const{
   Circle::print();
   cout << "; Height = " << height;}
```

```
// Driver for shape, point, circle, cylinder hierarchy
#include <iostream>
#include <iomanip>

#include "shape.h"
#include "point1.h"
#include "circle1.h"
#include "cylindr1.h"

void virtualViaPointer( const Shape * );
void virtualViaReference( const Shape & );

int main()
{
   cout << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 2 );

   Point point( 7, 11 );                  // create a Point
   Circle circle( 3.5, 22, 8 );           // create a Circle
   Cylinder cylinder( 10, 3.3, 10, 10 );  // create a Cylinder
...
```

```
   point.printShapeName();    // static binding
   point.print();             // static binding
   cout << '\n';

   circle.printShapeName();   // static binding
   circle.print();            // static binding
   cout << '\n';

   cylinder.printShapeName(); // static binding
   cylinder.print();          // static binding
   cout << "\n\n";

   Shape *arrayOfShapes[ 3 ];  // array of base-class pointers

   // aim arrayOfShapes[0] at derived-class Point object
   arrayOfShapes[ 0 ] = &point;

   // aim arrayOfShapes[1] at derived-class Circle object
   arrayOfShapes[ 1 ] = &circle;

   // aim arrayOfShapes[2] at derived-class Cylinder object
   arrayOfShapes[ 2 ] = &cylinder;
```

```
 // Loop through arrayOfShapes and call virtualViaPointer
   // to print the shape name, attributes, area, and volume
   // of each object using dynamic binding.
   cout << "Virtual function calls made off "
        << "base-class pointers\n";

   for ( int i = 0; i < 3; i++ )
      virtualViaPointer( arrayOfShapes[ i ] );

   // Loop through arrayOfShapes and call virtualViaReference
   // to print the shape name, attributes, area, and volume
   // of each object using dynamic binding.
   cout << "Virtual function calls made off "
        << "base-class references\n";

   for ( int j = 0; j < 3; j++ )
      virtualViaReference( *arrayOfShapes[ j ] );

   return 0;
}
```

```
// Make virtual function calls off a base-class pointer
// using dynamic binding.
void virtualViaPointer( const Shape *baseClassPtr )
{
   baseClassPtr->printShapeName();
   baseClassPtr->print();
   cout << "\nArea = " << baseClassPtr->area()
        << "\nVolume = " << baseClassPtr->volume() << "\n\n";
}

// Make virtual function calls off a base-class reference
// using dynamic binding.
void virtualViaReference( const Shape &baseClassRef )
{
   baseClassRef.printShapeName();
   baseClassRef.print();
   cout << "\nArea = " << baseClassRef.area()
        << "\nVolume = " << baseClassRef.volume() << "\n\n";
}
```