POLYMORPHISM

There are three major concepts in object-oriented programming:

1. Encapsulation (Classes),
   Data abstraction, information hiding (public: interface, private: implementation)
2. Inheritance,
   Is-a relation, reusability
3. Polymorphism
   Run-time decision for function calls (dynamic method binding)

In real life, there is often a collection of different objects that, given identical instructions (messages), should take different actions.

Take teacher and principal, for example.

Suppose the minister of education wants to send a directive to all personnel: "Print your personal information!".

Different kinds of staff (teacher or principal) have to print different information.

But the minister doesn't need to send a different message to each group. One message works for everyone because everyone knows how to print his or her personal information.

Besides the minister don't need to know **the type** of the person to whom the message is to be sent.

Polymorphism means "**taking many shapes**".

The minister's single instruction is polymorphic because it looks different to different kinds of personnel.

The minister doesn't need to know the type of the person to whom she/he sends the message.

Typically, polymorphism occurs in classes that are related by inheritance.

In C++, polymorphism means that a call to a member function will cause a different function to be executed depending on the **type of object** that gets the message.

The sender of the message don't need to know the type of the receiving object.

This sounds a little like function overloading, but polymorphism is a different, and much more powerful, mechanism.

One difference between overloading and polymorphism has to do with which function to execute when the choice is made.

With function overloading, the choice is made by the compiler (**compile-time**).

With polymorphism, it's made while the program is running (**run-time**).

Normal Member Functions Accessed with Pointers

The first example shows what happens when a base class and derived classes all have functions with the same name
and these functions are accessed using pointers but *without* using virtual functions (without polymorphism).

```
class Teacher{                                          // Base class
      string name;
      int numOfStudents;
  public:
      Teacher(const string &, int);                     // Constructor of base
      void print() const;
};

class Principal : public Teacher{                       // Derived class
      string SchoolName;
  public:
      Principal(const string &,  int , const string &);
      void print() const;
};
```

Both classes have a function with the same name: print. But in this example these functions are not virtual (**not polymorphic**).

```
// Show is a system that operates on Teachers and Principals
void show (const Teacher * tp)
{
   tp->print();       // which print
}


// Only to test the show function
int main()
{
   Teacher t1("Teacher 1", 50);
   Principal p1("Principal 1", 40, "School");
   Teacher *ptr;
   char c;
   cout << "Teacher or Principal " ; cin >> c;
   if (c == 't') ptr = &t1;
         else    ptr = &p1;
    show(ptr);                  // which print ??
     :
```

The Principal class is derived from class Teacher. Both classes have a member function with the same name print().

In main(), the program defines a pointer to class Teacher.

The main program sometimes puts the address of the **Teacher** object and sometimes the address of a derived class object (**Principal**) in the base class pointer as in the line

ptr = &p1;

Remember that it's all right to assign an address of derived type to a pointer of the base.

This address is sent to the function show() over a pointer to base.

Now the question is, when you execute the statement
tp->print();
what function is called? Is it Teacher::print() or Principal::print()?

See Example: e81.cpp

The function in the base class (Teacher) is executed in both cases. The compiler ignores the **contents** of the pointer ptr and chooses the member function that matches the **type** of the pointer.

Virtual Member Functions Accessed with Pointers

Let's make a single change in the program: Place the keyword **virtual** in front of the declaration of the print() function in the base class.

```
class Teacher{                                  // Base class
      string name;
      int numOfStudents;
   public:
      Teacher(const string &, int);             // Constructor of base
      virtual void print() const;               // A virtual (polymorphic) function
};

class Principal : public Teacher{               // Derived class
      string SchoolName;
   public:
      Principal(const string &,  int , const string &);
      void print() const;                       // It is also virtual (polymorphic)
};
```

See Example:e82.cpp

Now, different functions are executed, depending on the contents of ptr. Functions are called based on the **contents** of the pointer, not on the *type*.

This is polymorphism at work. I've made print() polymorphic by designating it virtual.

Benefits of Polymorphism

Polymorphism provides flexibility.

In our example the show() function has no information about the type of object pointed by the input parameter.

It can have the address of an object of any class derived from the Teacher class.

So this function can operate on any class derived from the Teacher.

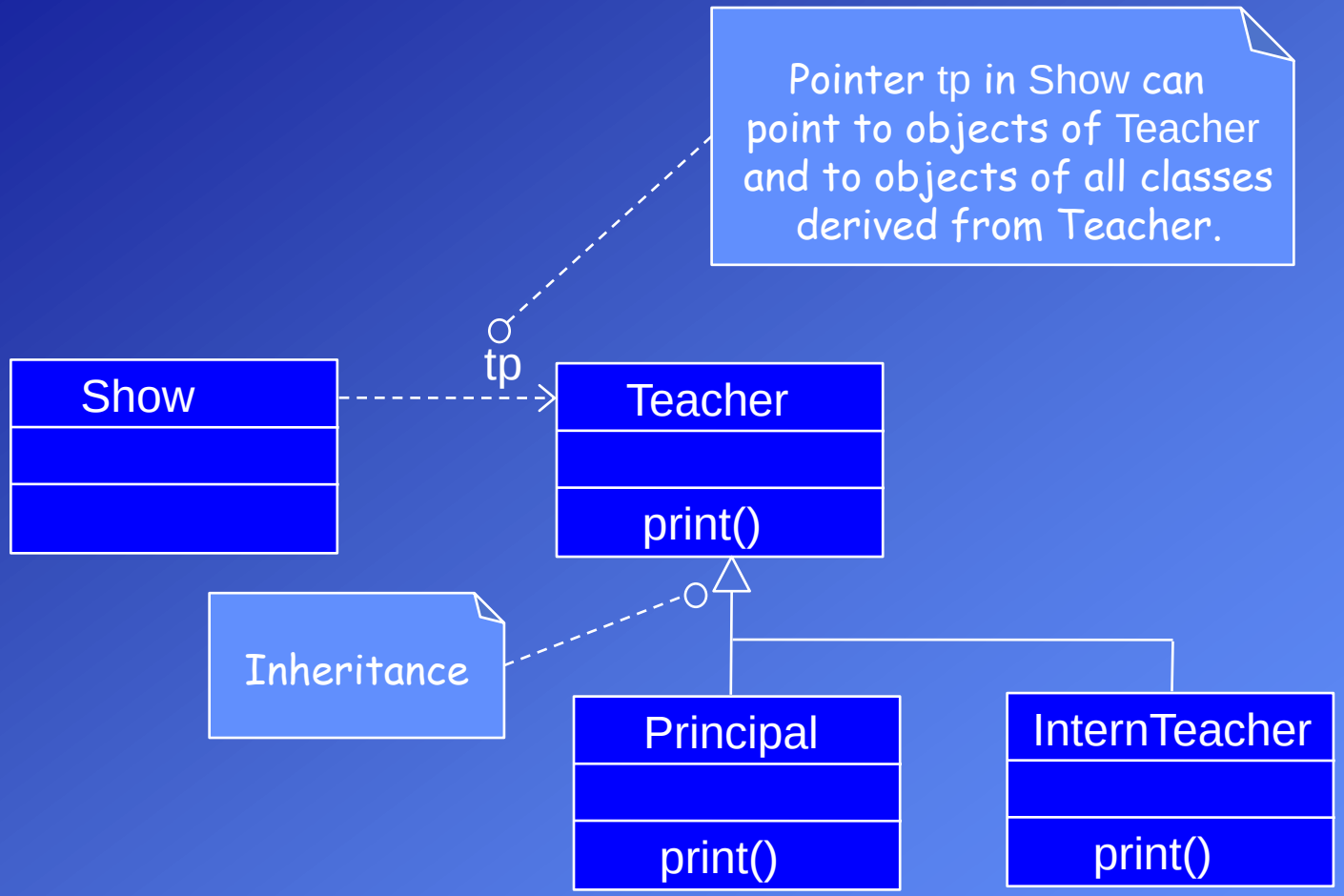
If we add a new teacher type (a new class) to the system, for example InternTeacher, we do not need to change the show function.

The same thing is true, if we discard a class derived from Teacher from the system.

UML Diagram of the Design

The Unified Modeling Language (UML) is explained in Chapter 9.

This is only a simple example.



Note that, in C++ it is preferred to use references instead of pointers by passing parameters.

The same program can be written as follows:

```
// Show is a system that operates on Teachers and Principals
void show (const Teacher & tp)
{
   tp.print();       // which print
}

// Only to test the show function
int main()
{
   Teacher t1("Teacher 1", 50);
   Principal p1("Principal 1", 40, "School");
   char c;
   cout << "Teacher or Principal " ; cin >> c;
   if (c == 't') show(t1);
         else    show(p1);
     :
```

Late Binding

How does the compiler know what function to compile? In e81.cpp, without polymorphism, the compiler has no problem with the expression tp->print();

It always compiles a call to the print() function in the base class.

But in e82.cpp, the compiler doesn't know what class the contents of tp may be a pointer to. It could be the address of an object of the Teacher class or the Principal class.

Which version of print() does the compiler call?

In fact, at the time it's compiling the program, the compiler doesn't "know" which function to call. So instead of a simple function call, it places a piece of code there.

At runtime, when the function call is executed, code that the compiler placed in the program finds out the type of the object whose address is in tp and calls the appropriate print() function: Teacher::print() or Principal::print().

Selecting a function at runtime is called **late binding** or **dynamic binding**. (Binding means connecting the function call to the function.)

Connecting to functions in the normal way, during compilation, is called *early binding* or *static binding*.

Late binding requires a small amount of overhead (the call to the function might take something like 10 percent longer) but provides an enormous increase in power and flexibility.

How It Works

Remember that, stored in memory, a normal object—that is, one with no virtual functions—contains only its own data, nothing else.

When a member function is called for such an object, the address of the object is available in the this pointer, which the member function uses (usually invisibly) to access the object's data.

The address in this is generated by the compiler every time a member function is called; it's not stored in the object.

With virtual functions, things are more complicated. When a derived class with virtual functions is specified, the compiler creates a table—an array—of function addresses called the **virtual table**.

In the example e82.cpp, the Teacher and Principal classes each have their own virtual table. There is an entry in each virtual table for every virtual function in the class.

Objects of classes with virtual functions contain a pointer to the virtual table (**vptr**) of the class. These object are slightly larger than normal objects.

In the example, when a virtual function is called for an object of Teacher or Principal, the compiler, instead of specifying what function will be called, creates code that will first look at the object's virtual table and then uses this to access the appropriate member function address.

Thus, for virtual functions, the object itself determines what function is called, rather than the compiler.

Example: *Assume that the classes* Teacher and Principal *contain two virtual functions.*

```
class Teacher{                                  // Base class
      string name;
      int numOfStudents;
  public:
      virtual void read();                      // Virtual function
      virtual void print() const;               // Virtual function
};

class Principal : public Teacher{               // Derived class
     string SchoolName;
  public:
     void read();                               // Virtual function
     void print() const;                        // Virtual function
};
```

Virtual Table of Teacher

&Teacher::read
&Teacher::print

Virtual Table of Principal

&Principal::read
&Principal::print

Objects of Teacher and Principal will contain a pointer to their virtual tables.

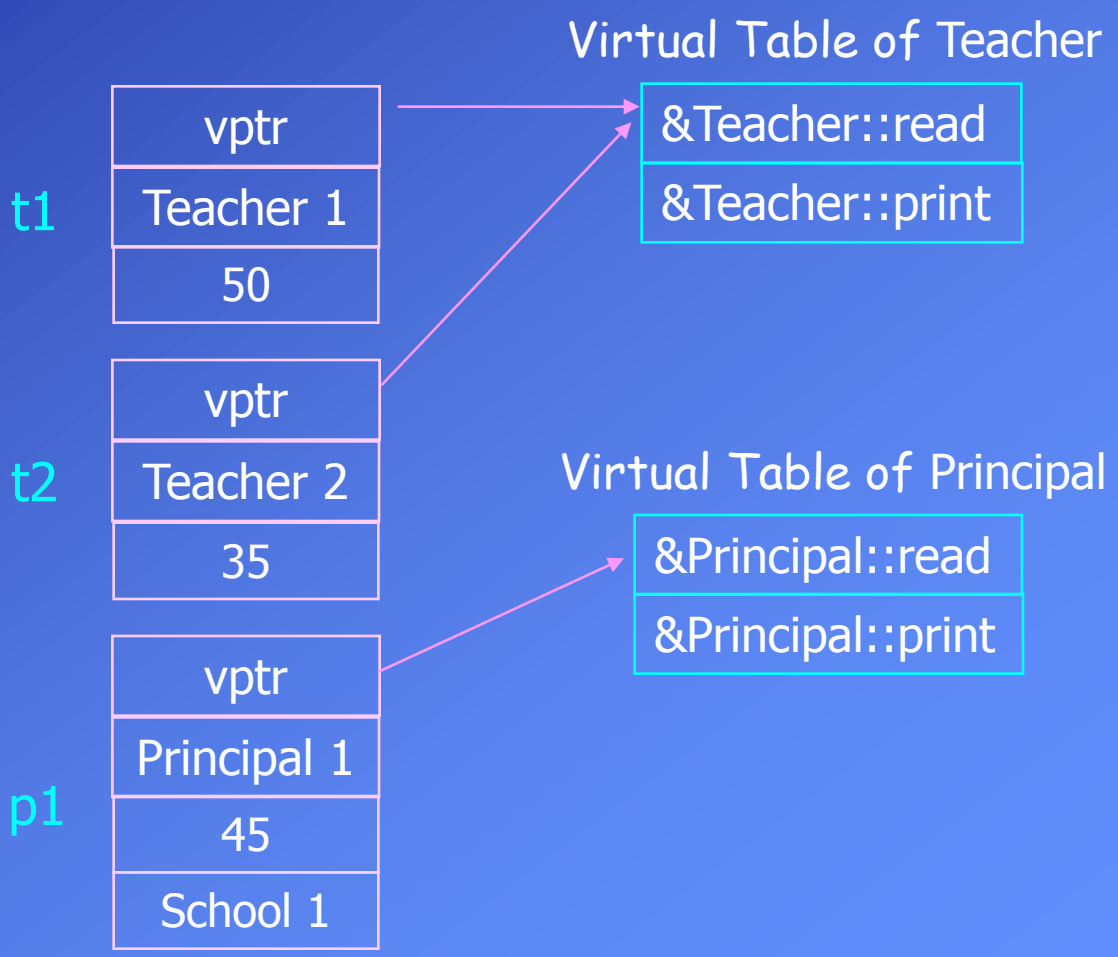
```
int main(){
   Teacher t1("Teacher 1", 50);
   Teacher t2("Teacher 2", 35);
   Principal p1("Principal 1", 45 , "School 1");
   :
}
```

MC68000-like assembly counterpart of the statement ptr->print(); Here ptr contains the address of an object.

```
move.l    ptr, this  ; this to object
movea.l   ptr, a0    ; a0 to object
movea.l   (a0), a1 ; a1<-vptr
jsr       4(a1)      ; jsr print
```

If the print() function would not a virtual function:

```
move.l    ptr, this  ; this to object
jsr       teacher_print
   or
jsr       principal_print
```



Don't Try This with Objects

Be aware that the virtual function mechanism works only with pointers to objects and, with references, not with objects themselves.

```
int main()
{
    Teacher t1("Teacher 1", 50);
    Principal p1("Principal 1", 40, "School");
    t1.print();                                  // not polymorphic
    p1.print();                                  // not polymorphic
    return 0;
}
```

Calling virtual functions is a time-consuming process, because of indirect call via tables. Don't declare functions as virtual if it is not necessary.

Linked List of Objects and Polymorphism

The most common ways to use virtual functions are with an array of pointers to objects and linked lists of objects.

Examine the example: See Example: e83.cpp

Remember, there is a **list** class in the standard library of the C++. You don't need to write a class to define linked lists.

Abstract Classes

To write polymorphic functions we need to have derived classes.

But sometimes we don't need to create any base class objects, but only derived class objects. The base class exists only as a starting point for deriving other classes.

This kind of base class is called an ***abstract class***, which means that no actual objects will be created from it.

Abstract classes arise in many situations. A factory can make a sports car or a truck or an ambulance, but it can't make a generic vehicle.

The factory must know the details about what *kind* of vehicle to make before it can actually make one.

Pure Virtual Functions

It would be nice if, having decided to create an abstract base class, I could instruct the compiler to *prevent* any class user from ever making an object of that class.

This would give me more freedom in designing the base class because I wouldn't need to plan for actual objects of the class, but only for data and functions that would be used by derived classes.

There is a way to tell the compiler that a class is abstract: You define at least one ***pure virtual function*** in the class.

A pure virtual function is a virtual function with no body. The body of the virtual function in the base class is removed, and the notation **=0** is added to the function declaration.

A Graphics Example

```
class GenericShape{                                  // Abstract base class
   protected:
       int x, y;
   public:
      GenericShape(int x_in, int y_in){ x = x_in; y = y_in; }   // Constructor
      virtual void draw() const =0;                              // pure virtual function
};

class Line:public GenericShape{        // Line class
 protected:
   int x2,y2;                           // End coordinates of line
 public:
   Line(int x_in,int y_in,int x2_in,int y2_in):GenericShape(x_in,y_in),  x2(x2_in),y2(y2_in)
   { }
   void draw()const;                    // virtual draw function
};
void Line::draw()const
{
   cout << "Type: Line" << endl;
   cout << "Coordinates of end points: " << "X1=" << x <<  " ,Y1=" << y <<
                                            " ,X2=" << x2 <<  " ,Y2=" << y2 << endl;
}
```

```
class Rectangle:public GenericShape{       // Rectangle class
 protected:
    int x2,y2;              // coordinates of 2nd corner point
 public:
    Rectangle(int x_in,int y_in,int x2_in,int y2_in):GenericShape(x_in,y_in),
                                                    x2(x2_in),y2(y2_in)
    { }
    void draw()const;           // virtual draw
};
void Rectangle::draw()const
{
    cout << "Type: Rectangle" << endl;
    cout << "Coordinates of corner points: " << "X1=" << x << " ,Y1=" << y <<
                                                  " ,X2=" << x2 <<  " ,Y2=" << y2 << endl;
}

class Circle:public GenericShape{        // Circle class
 protected:
    int radius;
 public:
    Circle(int x_cen,int y_cen,int r):GenericShape(x_cen,y_cen), radius(r)
    { }
    void draw() const;          // virtual draw
};
```

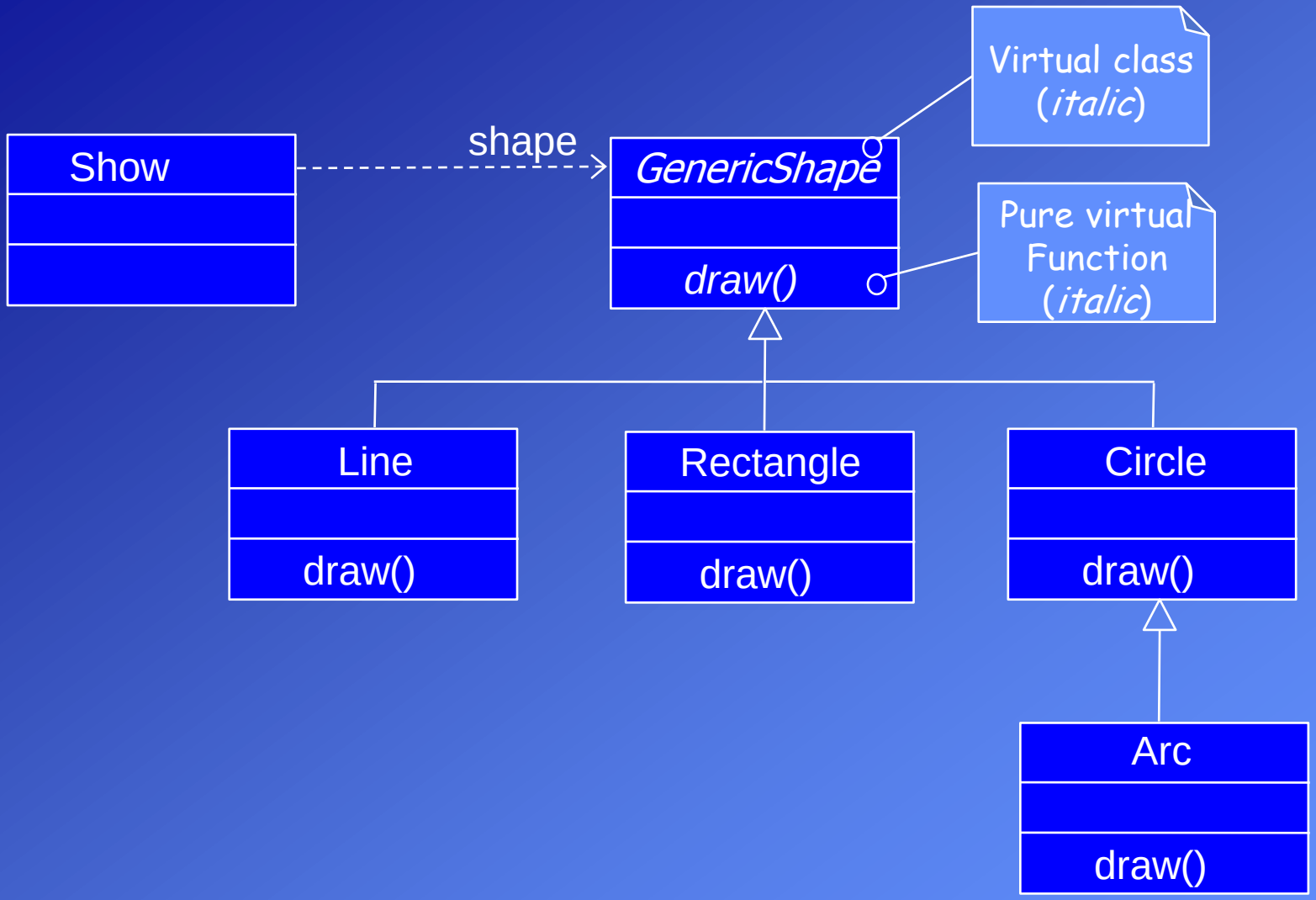
```
void Circle::draw()const
{
    cout << "Type: Circle" << endl;
    cout << "Coordinates of center point: " << "X=" << x <<  " ,Y=" << y << endl;
    cout << "Radius: " << radius << endl;
}

// A function to draw different shapes
void show(const Generic_shape &shape)  //Can take references to different shapes
{                                      // Which draw function will be called?
    shape.draw();                      // It 's unknown at compile-time
}

int main()          // A main function to test the system
{
    Line line1(1, 1, 100, 250);
    Circle circle1(100, 100, 20);
    Rectangle rectangle1(30, 50, 250, 140);
    Circle circle2(300, 170, 50);
    show(circle1);                  // show function can take different shapes as argument
    show(line1);
    show(circle2);
    show(rectangle1);
    return 0;
}
```

See Example: e84a.cpp

UML Diagram of the Design



If we write a class for a new shape by deriving it from an existing class, we don't need to modify the show function. This function can also show the new shape.

For example we can add an Arc class to our graphics library, which will not effect the show function.

```
class Arc:public Circle{                    // Arc class
 protected:
    int sa, ea;                             // Start and end angles
 public:
    Arc(int x_cen,int y_cen,int r, int a1, int a2):Circle(x_cen,y_cen,r),
                                                   sa(a1),ea(a2)
     {}
    void draw() const;                      // virtual draw
};

void Arc::draw()const
{
   cout << "Type: Arc" << endl;
   cout << "Coordinates of center point: " << "X=" << x <<  " ,Y=" << y << endl;
   cout << "Radius: " << radius << endl;
   cout << "Start and end angles: " << "SA=" << sa <<  " ,EA=" << ea << endl;
}
```

See Example: e84b.cpp

Virtual Constructors?

Can constructors be virtual?

No, they can't be.

When you're creating an object, you usually already know what kind of object you're creating and can specify this to the compiler. Thus, there's not a need for virtual constructors.

Also, an object's constructor sets up its virtual mechanism (the virtual table) in the first place. You don't see the code for this, of course, just as you don't see the code that allocates memory for an object.

Virtual functions can't even exist until the constructor has finished its job, so **constructors can't be virtual**.

Virtual Destructors

See Example: e85.cpp

Recall that a derived class object typically contains data from both the base class and the derived class. To ensure that such data is properly disposed of, it may be essential that destructors for both base and derived classes are called. But the output of e85.cpp is

Base Destructor
Program terminates

In this program bp is a pointer of Base type. So it can point to objects of Base type and Derived type. In the example, bp points to an object of Derived class, but while deleting the pointer only the Base class destructor is called.

This is the same problem you saw before with ordinary (nondestructor) functions.

If a function isn't virtual, only the base class version of the function will be called when it's invoked using a base class pointer, even if the contents of the pointer is the address of a derived class object.

Thus in e85.cpp, the Derived class destructor is never called. This could be a problem if this destructor did something important.

To fix this problem, we have to make the base class **destructor virtual**.

Rewrite: e85.cpp