Virtual Functions

CHAPTER 12

Normal member function accessed with Pointers

NOTVIRT

Base Class

```
class Base //base class
{
public:
void show() //normal function
{
 cout << “Base\n”;
}
};
```

Derived Class 1

```
class Derv1 : public Base //derived class 1
{
public:
void show()
{
cout << “Derv1\n”;
}
};
```

Derived Class 2

```
class Derv2 : public Base //derived class 2
{
public:
void show()
{
cout << “Derv2\n”;
}
};
```

Main

```
int main()
{
Derv1 dv1; //object of derived class 1
Derv2 dv2; //object of derived class 2
Base* ptr; //pointer to base class
ptr = &dv1; //put address of dv1 in pointer
ptr->show(); //execute show()
ptr = &dv2; //put address of dv2 in pointer
ptr->show(); //execute show()
return 0;
}
```

ptr = &dv1; // derived class address in base class pointer

How can we assign an address of one type (Derv1) to a pointer of another (Base)?

Derived class address in Base class Pointer

The rule is that pointers to objects of a derived class are type compatible with pointers to objects of the base class

ptr->show();

which function is called? Is it

Base::show()

or

Derv1::show()?

Which function is called?

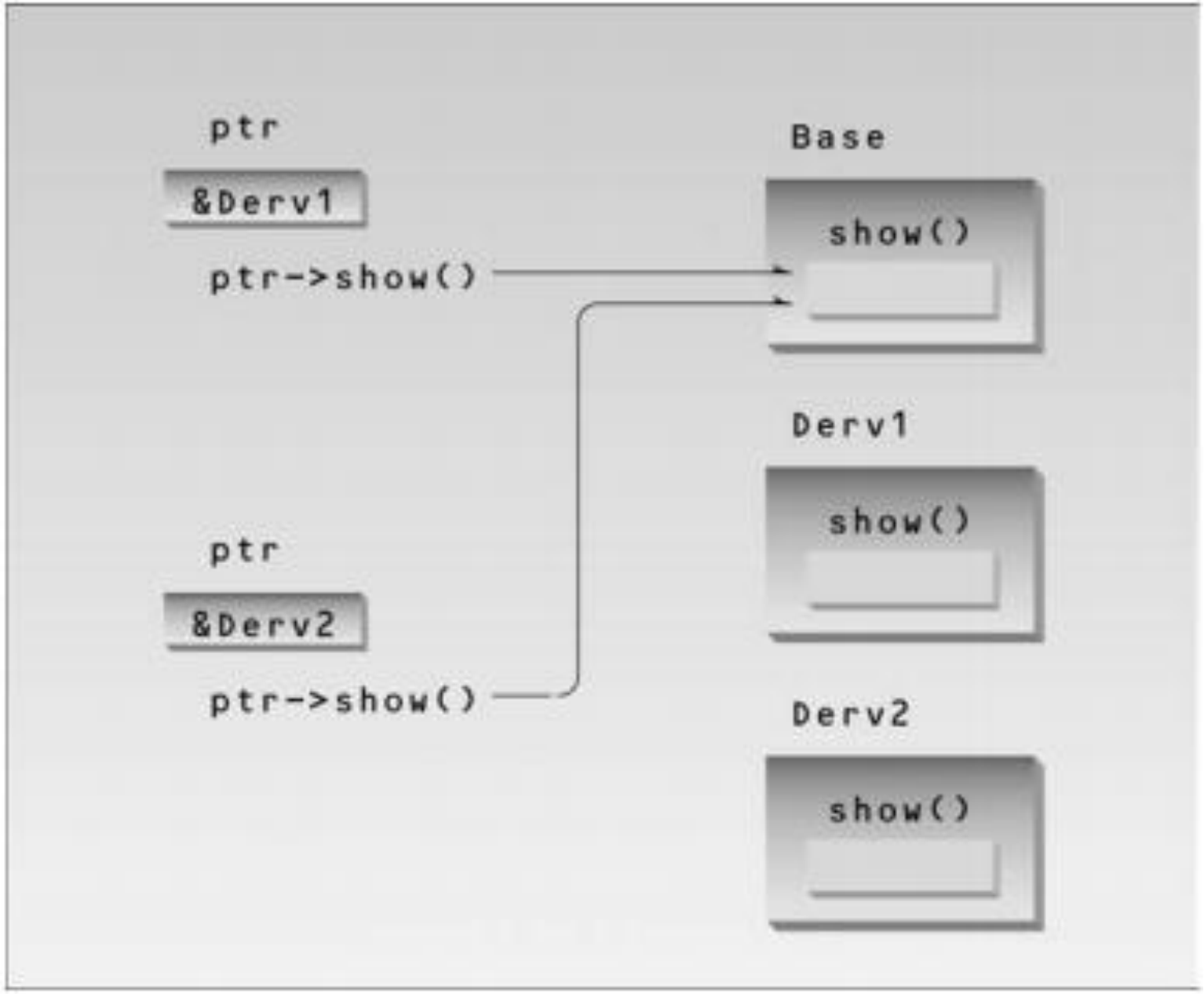
the function in the base class is executed.

The compiler ignores the contents of the pointer ptr and chooses the member function that matches the type of the pointer

Output

Base

Base


ptr
&Derv1
ptr->show()
Base
show()
Derv1
show()
ptr
&Derv2
ptr->show()
Derv2
show()

Virtual member function accessed with Pointers

VIRT

Base Class

```
class Base //base class
{
public:
virtual void show() //virtual function
{
cout << “Base\n”;
}
};
```

Derived Class 1

```
class Derv1 : public Base //derived class 1
{
public:
void show()
{
cout << “Derv1\n”;
}
};
```

Derived Class 2

```
class Derv2 : public Base //derived class 2
{
public:
void show()
{
cout << “Derv2\n”;
}
};
```

Main

```
int main()
{
Derv1 dv1; //object of derived class 1
Derv2 dv2; //object of derived class 2
Base* ptr; //pointer to base class
ptr = &dv1; //put address of dv1 in pointer
ptr->show(); //execute show()
ptr = &dv2; //put address of dv2 in pointer
ptr->show(); //execute show()
return 0;
}
```

Output

Derv1

Derv2

Function Call

Now, as you can see, the member functions of the derived classes, not the base class, are executed.

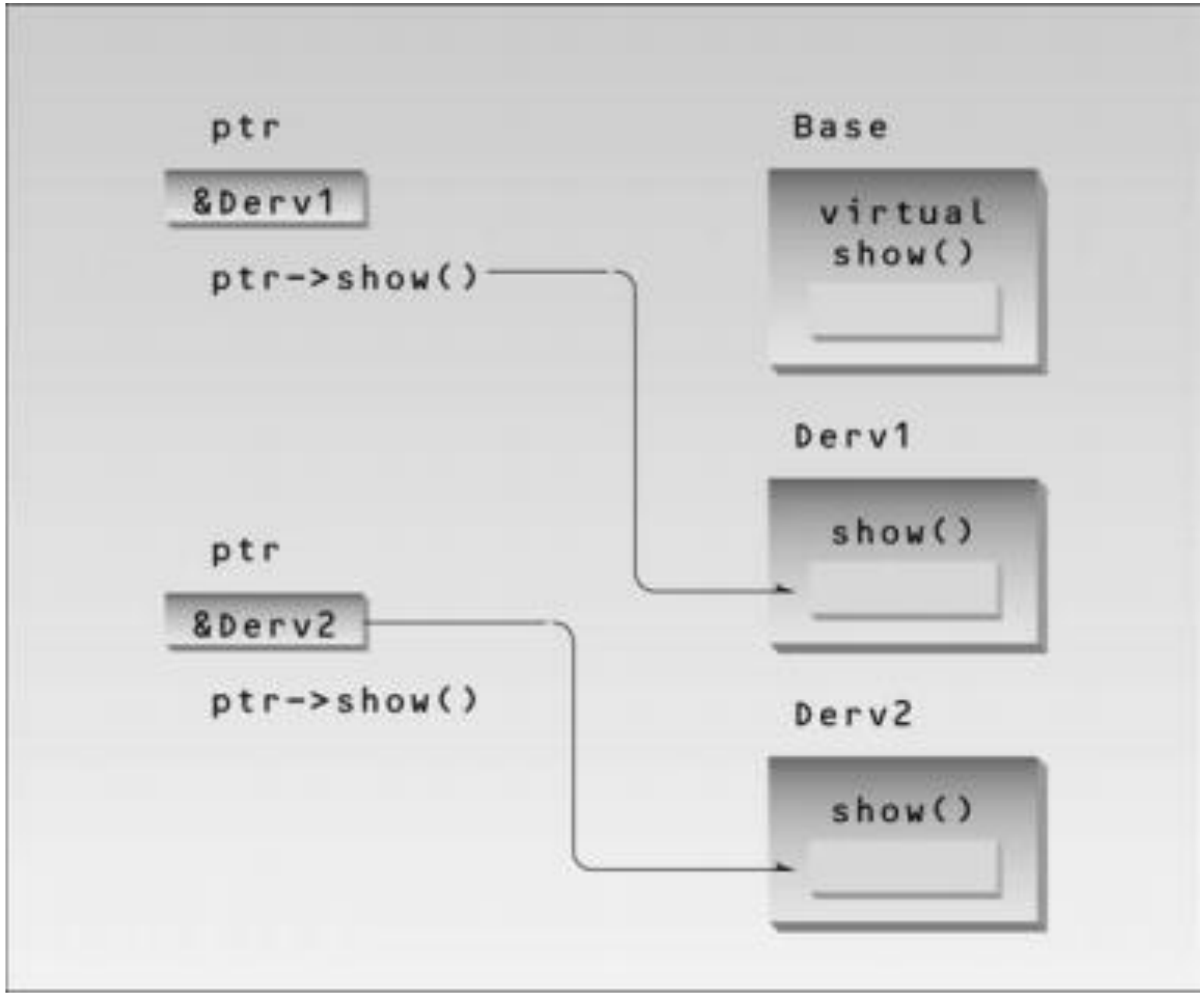
We change the contents of ptr from the address of Derv1 to that of Derv2, and the particular instance of show() that is executed also changes.

So the same function call

```
ptr->show();
```

executes different functions, depending on the contents of ptr.

The rule is that the compiler selects the function based on the contents of the pointer ptr, not on the type of the pointer


ptr
&Derv1
ptr->show()
Base
virtual
show()
Derv1
show()
ptr
&Derv2
ptr->show()
Derv2
show()

Late Binding

Late Binding

In NOTVIRT the compiler has no problem with the expression

ptr->show();

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

But in VIRT the compiler doesn’t know what class the contents of ptr may contain.

It could be the address of an object of the Derv1 class or of the Derv2 class.

Which version of show() does the compiler call?

Late Binding

In fact the compiler doesn’t know what to do, so it arranges for the decision to be deferred until the program is running.

At runtime, when it is known what class is pointed to by ptr, the appropriate version of draw will be called.

This is called late binding or dynamic binding.

Choosing functions in the normal way, during compilation, is called early binding or

static binding.

Late binding requires some overhead but provides increased power and flexibility.

Virtual Functions

- Virtual means existing in appearance but not in reality
- When virtual functions are used a program that appears to be calling a function of one class may be calling a function of another class in reality.
- A virtual function is a member function which is declared within base class and is re-defined (Overriden) by derived class.
- When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Virtual Functions

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at Run-time.

Abstract Classes & Pure Virtual Functions

Abstract class

Suppose we have a base class and we want to restrict programmers using our library from making objects of the base class

How can we make it clear to someone using our family of classes that we don't want anyone to instantiate objects of the base class?

We could just say this in the documentation, and count on the users of the class to remember it, but of course it's much better to write our classes so that such instantiation is impossible.

How can we can do that?

By placing at least one pure virtual function in the base class

If a class contains at least one pure virtual function it is called abstract.

Such a class exists only to act as a parent of derived classes that will be used to instantiate objects.

It may also provide an interface for the class hierarchy.

Pure Virtual Function

A pure virtual function is one with the expression =0 added to the declaration

```
virtual void show() = 0; // pure virtual function
```

The equal sign here has nothing to do with assignment; the value 0 is not assigned to anything.

The =0 syntax is simply how we tell the compiler that a virtual function will be pure.

Now if in main() you attempt to create objects of class Base, the compiler will complain that you're trying to instantiate an object of an abstract class.

It will also tell you the name of the pure virtual function that makes it an abstract class.

Notice that, although this is only a declaration, you never need to write a definition of the base class show(), although you can if you need to

Pure Virtual Function(Cont.)

Once you've placed a pure virtual function in the base class, you must override it in all the

derived classes from which you want to instantiate objects.

If a class doesn't override the pure virtual function, it becomes an abstract class itself, and you can't instantiate objects from it (although you might from classes derived from it).

For consistency, you may want to make all the virtual functions in the base class pure.

Base Class

```
class Base //base class
{
public:
virtual void show() = 0; //pure virtual function
};
```

Derived Class 1

```
class Derv1 : public Base //derived class 1
{
public:
void show()
{
cout << “Derv1\n”;
}
};
```

Derived Class 2

```
class Derv2 : public Base //derived class 2
{
public:
void show()
{
cout << “Derv2\n”;
}
};
```

Main

```
int main()
{
// Base bad; //can’t make object from abstract class
Base* arr[2]; //array of pointers to base class
Derv1 dv1; //object of derived class 1
Derv2 dv2; //object of derived class 2
arr[0] = &dv1; //put address of dv1 in array
arr[1] = &dv2; //put address of dv2 in array
arr[0]->show(); //execute show() in both objects
arr[1]->show();
return 0;
}
```

Output

Derv1

Derv2

Virtual Functions in Person Class

Person Class

```
class person //person class
{
protected:
char name[40];
public:
void getName()
{
cout << " Enter name: "; cin >> name;
}
void putName()
{
cout << "Name is: " << name << endl;
}
virtual void getData() = 0; //pure virtual func
virtual bool isOutstanding() = 0; //pure virtual func
};
```

Student Class

```
class student : public person //student class
{
private:
float gpa; //grade point average
public:
void getData() //get student data from user
{
person::getName();
cout << " Enter student's GPA: "; cin >> gpa;
}
bool isOutstanding()
{ return (gpa > 3.5) ? true : false; }
};
```

Professor Class

```
class professor : public person //professor class
{
private:
int numPubs; //number of papers published
public:
void getData() //get professor data from user
{
person::getName();
cout << " Enter number of professor's publications: ";
cin >> numPubs;
}
bool isOutstanding()
{
return (numPubs > 100) ? true : false;
}
};
```

Main

```
int main()
{
person* persPtr[100]; //array of pointers to persons
int n = 0; //number of persons on list
char choice;
```

Main(Cont.)

```
do {
cout << “Enter student or professor (s/p): “;
cin >> choice;
if(choice==’s’) //put new student
{persPtr[n] = new student;} // in array
else //put new professor
{persPtr[n] = new professor;} // in array
persPtr[n++]->getData(); //get data for person
cout << “ Enter another (y/n)? “; //do another person?
cin >> choice;
} while( choice==’y’ ); //cycle until not ‘y’
```

Main(Cont.)

```
for(int j=0; j<n; j++) //print names of all
{ //persons, and
persPtr[j]->putName(); //say if outstanding
if( persPtr[j]->isOutstanding() )
cout << " This person is outstanding\n";
}
return 0;
} //end main()
```

```
Enter student or professor (s/p): s
   Enter name: Timmy
   Enter student's GPA: 1.2
   Enter another (y/n)? y
Enter student or professor (s/p): s
   Enter name: Brenda
   Enter student's GPA: 3.9
   Enter another (y/n)? y
Enter student or professor (s/p): s
   Enter name: Sandy
   Enter student's GPA: 2.4
   Enter another (y/n)? y
Enter student or professor (s/p): p
   Enter name: Shipley
   Enter number of professor's publications: 714
   Enter another (y/n)? y
Enter student or professor (s/p): p
   Enter name: Wainright
   Enter number of professor's publications: 13
   Enter another (y/n)? n
```

```
Name is: Timmy
Name is: Brenda
    This person is outstanding
Name is: Sandy
Name is: Shipley
    This person is outstanding
Name is: Wainright
```

Polymorphism

Polymorphism

Using operators or functions in different ways, depending on what they are operating on, is called polymorphism (one thing with several distinct forms).

Overloading is a kind of polymorphism; it is also an important feature of OOP

Virtual Functions in Graphics Example

Shape Class

```
class shape //base class
{
protected:
int xCo, yCo; //coordinates of center
color fillcolor; //color
fstyle fillstyle; //fill pattern
public: //no-arg constructor
shape() : xCo(0), yCo(0), fillcolor(cWHITE),
fillstyle(SOLID_FILL)
{ } //4-arg constructor
shape(int x, int y, color fc, fstyle fs) :
xCo(x), yCo(y), fillcolor(fc), fillstyle(fs)
{ }
virtual void draw()=0 //pure virtual draw function
{
set_color(fillcolor);
set_fill_style(fillstyle);
}
};
```

Ball Class

```
class ball : public shape
{
private:
int radius; //(xCo, yCo) is center
public:
ball() : shape() //no-arg constr
{ }
//5-arg constructor
ball(int x, int y, int r, color fc, fstyle fs)
: shape(x, y, fc, fs), radius(r)
{ }
void draw() //draw the ball
{
shape::draw();
draw_circle(xCo, yCo, radius);
}
};
```

Rectangle Class

```
class rect : public shape
{
private:
int width, height; //(xCo, yCo) is upper left corner
public:
rect() : shape(), height(0), width(0) //no-arg ctor
{ } //6-arg ctor
rect(int x, int y, int h, int w, color fc, fstyle fs) :
shape(x, y, fc, fs), height(h), width(w)
{ }
void draw() //draw the rectangle
{
shape::draw();
draw_rectangle(xCo, yCo, xCo+width,
yCo+height);
set_color(cWHITE); //draw diagonal
draw_line(xCo, yCo, xCo+width, yCo+height);
}
};
```

Triangle Class

```
class tria : public shape
{
private:
int height; //(xCo, yCo) is tip of pyramid
public:
tria() : shape(), height(0) //no-arg constructor
{ } //5-arg constructor
tria(int x, int y, int h, color fc, fstyle fs) :
shape(x, y, fc, fs), height(h)
{ }
void draw() //draw the triangle
{
shape::draw();
draw_pyramid(xCo, yCo, height);
}
};
```

Main

```
int main()
{
int j;
init_graphics(); //initialize graphics system
shape* pShapes[3]; //array of pointers to shapes
//define three shapes
pShapes[0] = new ball(40, 12, 5, cBLUE, X_FILL);
pShapes[1] = new rect(12, 7, 10, 15, cRED, SOLID_FILL);
pShapes[2] = new tria(60, 7, 11, cGREEN, MEDIUM_FILL);
```

Main(Cont.)

```
for(j=0; j<3; j++) //draw all shapes
pShapes[j]->draw();
for(j=0; j<3; j++) //delete all shapes
delete pShapes[j];
set_cursor_pos(1, 25);
return 0;
}
```