Object-oriented Programming
Polymorphism

YoungWoon Cha

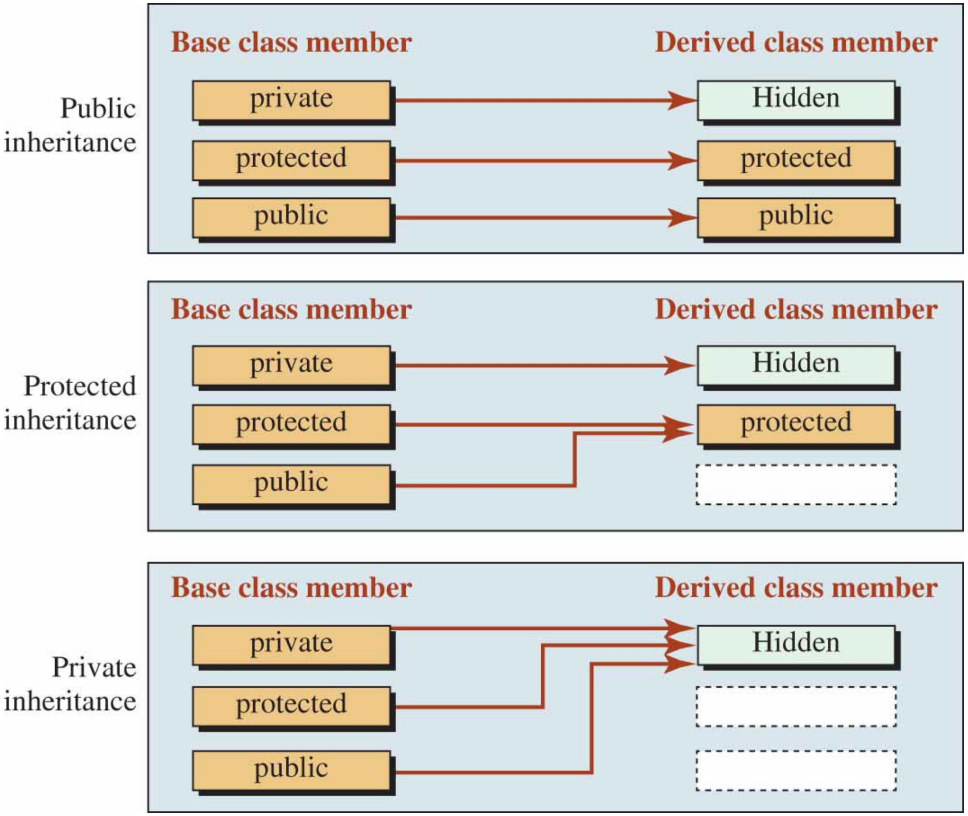
Computer Science and Engineering

Review

Three Types of Inheritance

Although, public inheritance is by far the most common type of derivation, C++ allows us to use two other types of derivation: *private* and *protected*.

Inheritance types

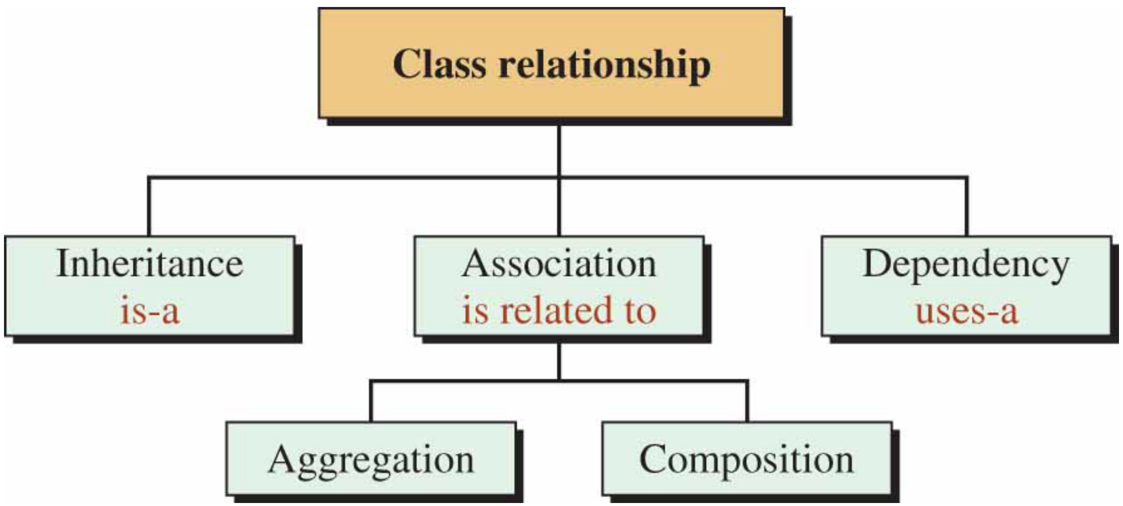


Class Relationships

In object-oriented programming, classes are used in relation to each other.

A program normally uses several classes with different relationships between them.

Relationship between classes



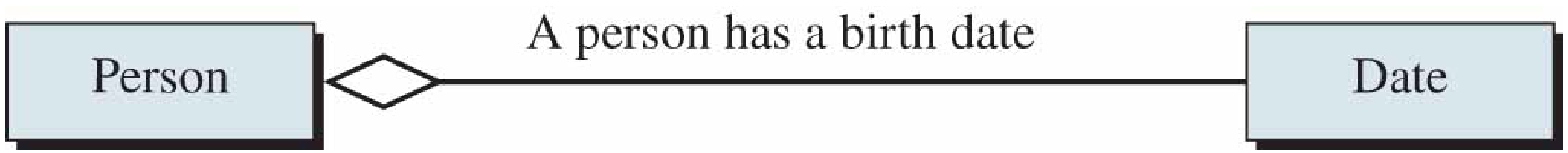
Has-a

Association

An aggregation is a one-to-many relationship from the aggregator to the aggregatee.

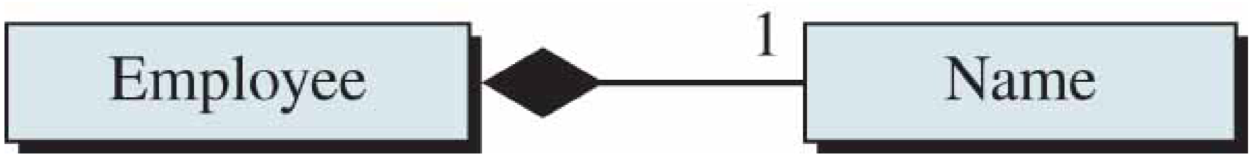
In an aggregation, the lifetime of the aggregatee is independent of the lifetime of the aggregator.

Example of aggregation relationship



The aggregatee must always belong to an aggregator and cannot have a life of its own.

Example of composition relationship

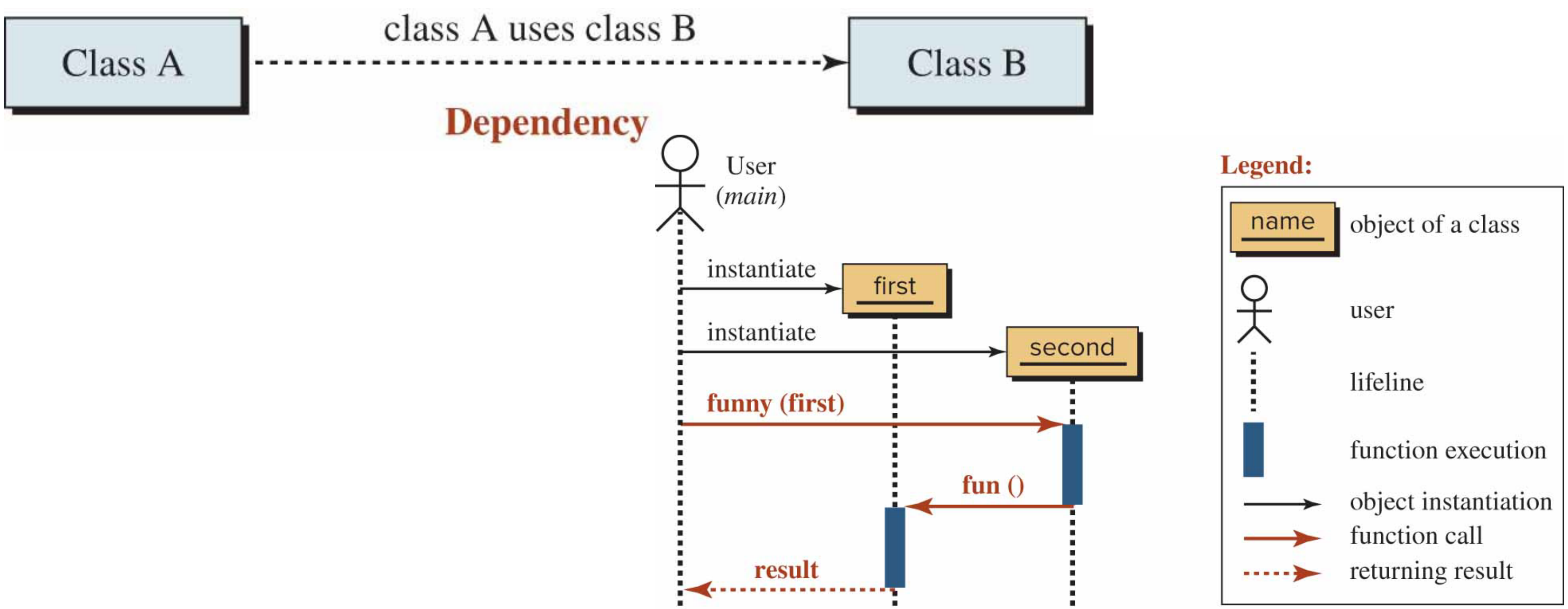


DEPENDENCY

Class A depends on class B if class A somehow uses class B. This happens when

- **Class A uses an object of type B as a parameter in a member function.**
- **Class A has a member function that returns an object of type B.**
- **Class A has a member function that has a local variable of type B.**

We use both UML class diagrams and UML sequence diagrams to show the dependencies.



Polymorphism in Inheritance

POLYMORPHISM

One of the main pillars of object-oriented programming is polymorphism.

Polymorphism gives us the ability to write several versions of a function, each in a separate class.

Then, when we call the function, the version appropriate for the object being referenced is executed.

Conditions for Polymorphism

We define the conditions for polymorphism in inheritance.

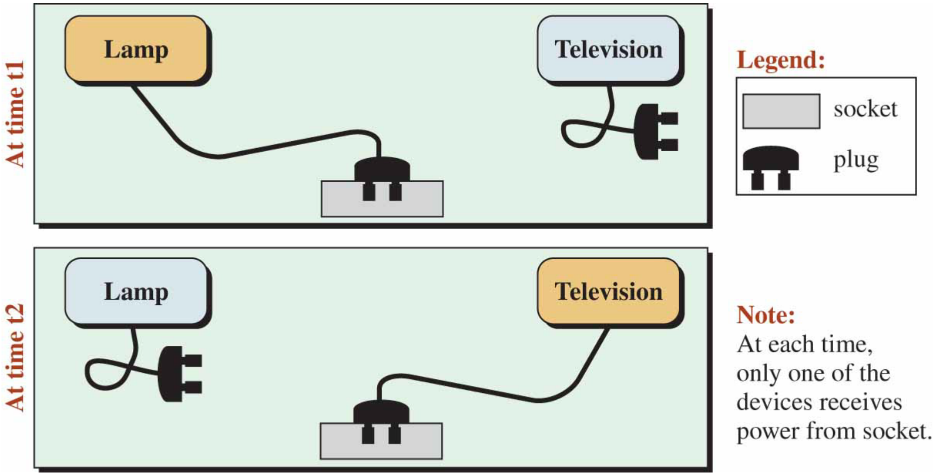
1. Pointer or References

In C++, a pointer or a reference can play the role of a socket that once created can accept plug-compatible objects.

We can define a pointer (or a reference) that can point to the base class; we can then let the pointer point to any object in the hierarchy.

For this reason, the pointer and reference variables are sometimes referred to as *polymorphic variables*.

A socket and plug-compatible devices



Conditions for Polymorphism

2. Exchangeable Objects

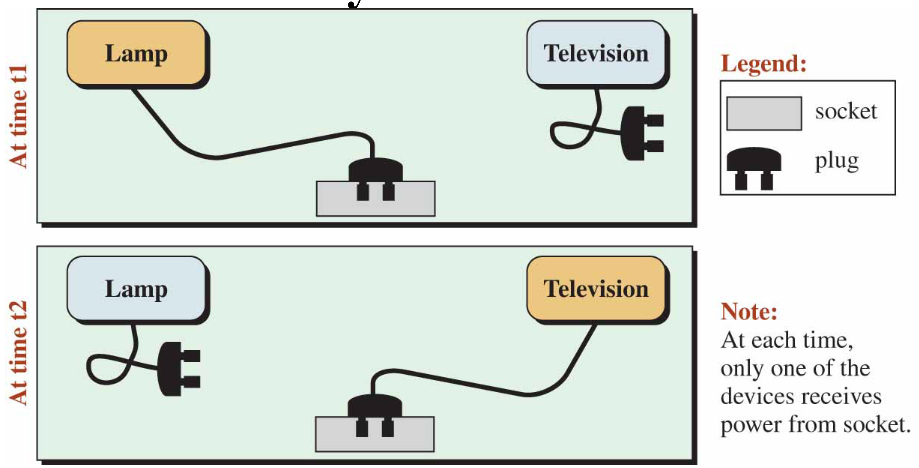
An object in an inheritance hierarchy plays the role of plug-compatible objects.

3. Virtual Functions

We need something to play the role of power given to different devices that perform different tasks.

This is done in C++ using virtual functions, that are modified by the keyword *virtual*.

For example, we can have a print function in all classes, all named the same but each prints differently.



Code Example #1

For polymorphism, we need pointers (or references), we need exchangeable objects, and we need virtual functions.

The following program shows the idea of polymorphism with only one pointer that can point to different objects.

However, the program shows an incomplete polymorphic program using only the first two conditions.

We define two classes. We then create a pointer (simulating a socket) that can accept an object of each class at different times (plug-compatible object).

For simplicity, we define only one public member function for each class and let the system add default constructors.

Incomplete Polymorphic Program Part 1

```
/***************************************************************
 * A simple program to show the first two conditions for       *
 * polymorphism                                                *
 ***************************************************************/
#include <iostream>
#include <string>
using namespace std;

// Definition of Base class and in-line print function
class Base
{
    public:
        void print () const {cout << "In the Base" << endl;}
};
// Definition of Derived class and in-line print function
class Derived : public Base
{
    public:
        void print () const {cout << "In the Derive" << endl;}
};
```

Incomplete Polymorphic Program Part 2

```
int main ( )
{
    // Creation of a pointer to the Base class (simulating socket)
    Base* ptr;
    // Let ptr points to an object of the Base class
    ptr = new Base ();
    ptr -> print();
    delete ptr;
    // Let ptr points to an object of the Derived class
    ptr = new Derived();
    ptr -> print();
    delete ptr;
    return 0;
}
```

```
Run:
In the Base
In the Base
```

Incomplete Polymorphic Program Part 3

We tried to call the function defined in the *Derived* class, but the result shows the function defined for the *Base* class is called.

The reason is that while the first two conditions of polymorphism are accomplished in the program, the third condition (virtual functions) is not fulfilled. (The print function is not a virtual function.)

```
// Creation of a pointer to the Base class (simulating socket)
Base* ptr;
// Let ptr points to an object of the Base class
ptr = new Base ();
ptr -> print();
delete ptr;
// Let ptr points to an object of the Derived class
ptr = new Derived();
ptr -> print();
delete ptr;
return 0;
```

The result should be expected because the variable *ptr* is defined as pointer to *Base*.

It can accept being pointed to an object of the *Derived* type because a derived object is a *Base* (inheritance defines an *is-a* relationship).

However, when it wants to call the print function, it is still a pointer to *Base*, so it calls the print function defined in the *Base* class.

We have not changed the type of the pointer, we have just forced it to point to the *Derived* class.

Code Example #2

For polymorphism, we need pointers (or references), we need exchangeable objects, and we need virtual functions.

We repeat the previous example, but we make the print function virtual.

The result is that the correct function is activated in each call as shown in the following Program.

Polymorphic Program Using All Three Conditions 1

```
/*****************************************************************
 * A simple program to show that if all three necessary         *
 * conditions are fulfilled, we have polymorphism               *
 *****************************************************************/
#include <iostream>
#include <string>
using namespace std;

// Definition of Base class and in-line definition for print function
class Base
{
    public:
        virtual void print () const {cout << "In the Base" << endl;}
};
// Definition of Derived class and in-line definition for print function
class Derived : public Base
{
    public:
        virtual void print () const {cout << "In the Derive" << endl;}
};
```

Polymorphic Program Using All Three Conditions 2

```
int main ( )
{
     // Creation of a pointer to the Base class (simulating socket)
     Base* ptr;
     // Let ptr points to an object of the Base class
     ptr = new Base ();
     ptr -> print();
     delete ptr;
     // Let ptr points to an object of the Derived class
     ptr = new Derived();
     ptr -> print();
     delete ptr;
     return 0;
}
```

```
Run:
In the Base
In the Derived
```

Need of Virtual Modifier

Although we have added the virtual modifier to both print functions, it is not needed.

When a function is defined as virtual, all functions in the hierarchy of classes with the same signature are automatically virtual.

```
/******************************************************************
 * A simple program to show that if all three necessary        *
 * conditions are fulfilled, we have polymorphism              *
 ******************************************************************/
#include <iostream>
#include <string>
using namespace std;

// Definition of Base class and in-line definition for print function
class Base
{
     public:
         virtual void print () const {cout << "In the Base" << endl;}
};
// Definition of Derived class and in-line definition for print function
class Derived : public Base
{
     public:
         void print () const {cout << "In the Derive" << endl;}
};
```

Mechanism

To understand how virtual functions take part in polymorphic behavior, we need to understand virtual tables (*vtables*).

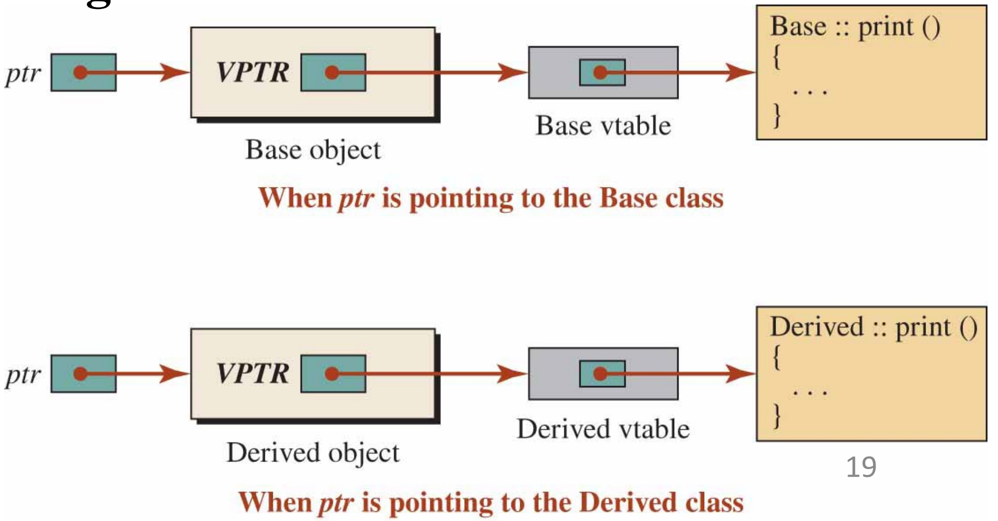
In polymorphism, the system creates a virtual table for each class in the hierarchy of classes.

Each entry in each *vtable* has a pointer to the corresponding virtual function.

Each object created in an application will have an extra data member (*VPTR*) which is a pointer to the corresponding *vtable*.

In the case of our simple program, there are two objects and two *vtables* each with one entry as shown in the Figure.

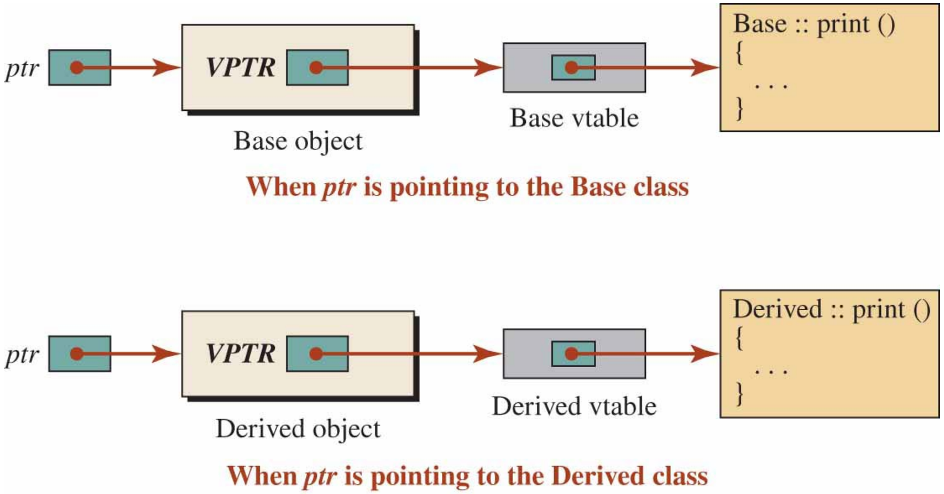
```
// Creation of a pointer to the Base class (simulating socket)
Base* ptr;
// Let ptr points to an object of the Base class
ptr = new Base ();
ptr -> print();
delete ptr;
// Let ptr points to an object of the Derived class
ptr = new Derived();
ptr -> print();
delete ptr;
return 0;
```



Virtual Tables for Base and Derived Classes

Virtual tables for Base and Derived classes

```
// Creation of a pointer to the Base class (simulating socket)
Base* ptr;
// Let ptr points to an object of the Base class
ptr = new Base ();
ptr -> print();
delete ptr;
// Let ptr points to an object of the Derived class
ptr = new Derived();
ptr -> print();
delete ptr;
return 0;
```



When ptr is pointing to the *Base* object, the *VPTR* pointer added to the *Base* class object is reached, which is pointing to the *vtable* of the *Base* class.

In this case, the *vtable* has only one entry which invokes the only virtual function in *Base* class.

When *ptr* is pointing to the *Derived* object, the *vtable* of the *Derived* object is reached and the print function defined in the *Derived* class is invoked.

Constructors and Destructors

Constructors and destructors in a class hierarchy are also member functions although special ones. We discuss them separately.

Constructor Cannot Be Virtual

Constructors cannot be virtual because although constructors are member function, the names of the constructors are different for the base and derived classes (different signatures).

Virtual Destructor

Although the name of the destructors are also different in the base and derived class, the destructors are normally not called by their name.

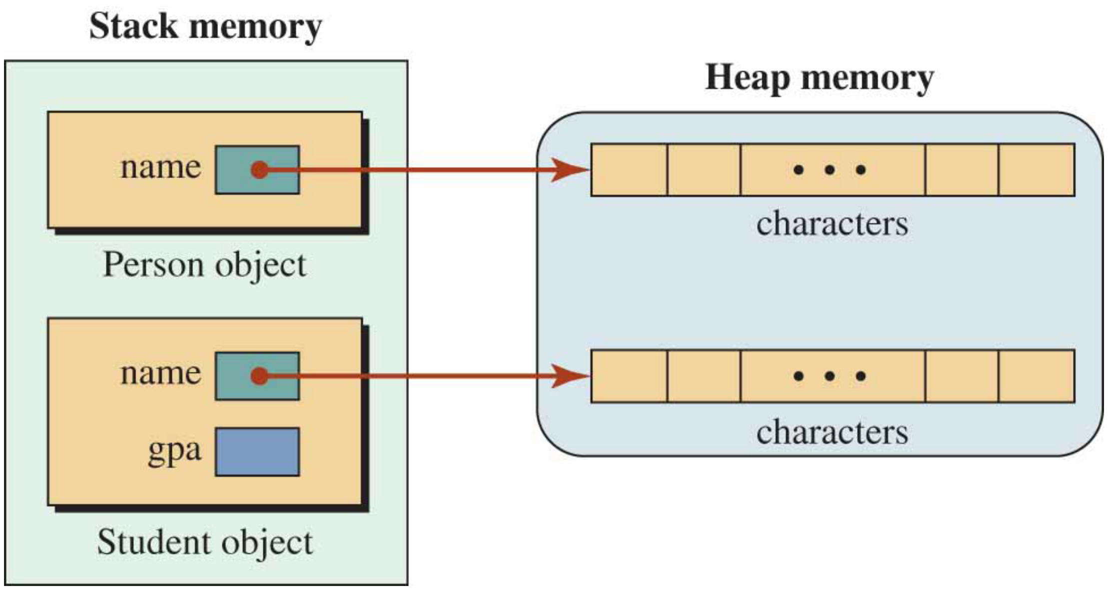
When there is a virtual member function anywhere in the design, we should make the destructors also virtual to avoid memory leaks.

Case 1: No Polymorphism

When we are not using polymorphism.

We create a *Person* class and a *Student* class.

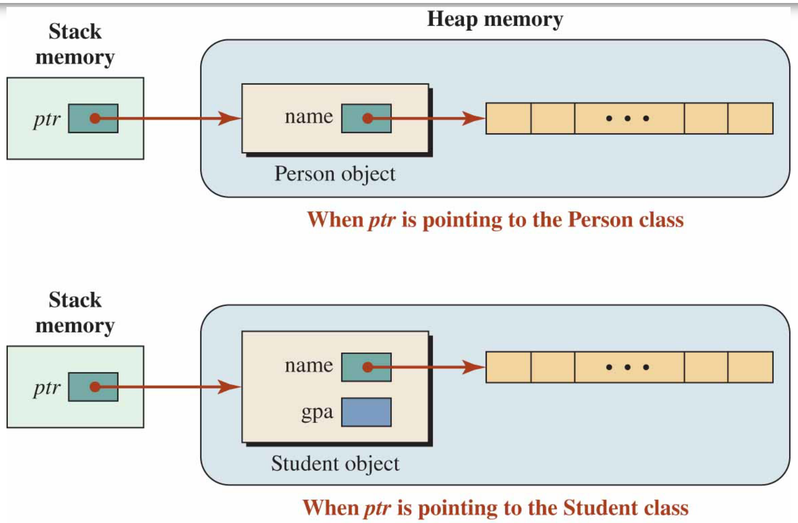
Two objects in a program not using polymorphism



We cannot have a memory leak in this situation.

When the program terminates the destructor for Person class and Student class are called, which automatically calls the destructors of the string class, which delete the allocated memory in the heap.

Case 2: Polymorphism Part 1



A problem may be created when ptr is deleted as shown below:

```
ptr = new Person(...);
...
delete ptr;  // It deletes Person because ptr type is Person*

ptr = new Student(...);
...
delete ptr; // It does not deletes Student because ptr type is Person*
```

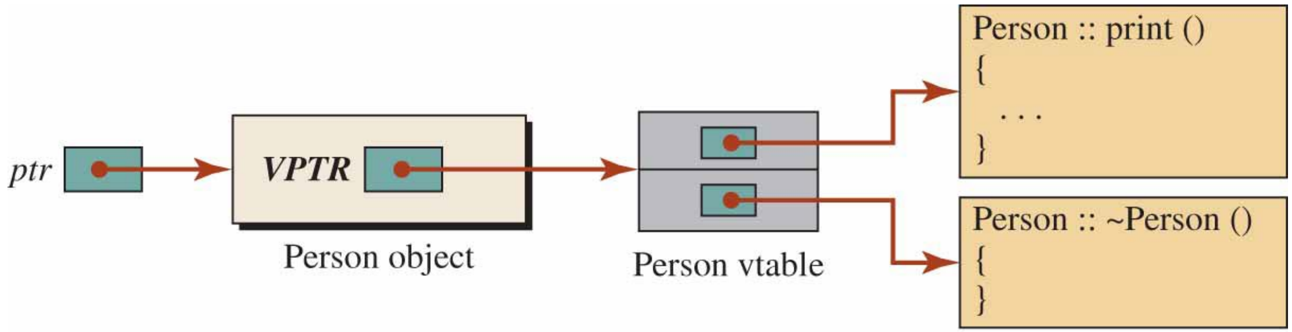
The solution is to make the destructor of the base class *virtual*, which automatically makes the destructor of the derived class *virtual*.

In this case, the system allows two different members functions with different names to be virtual and they are both added to the virtual table.

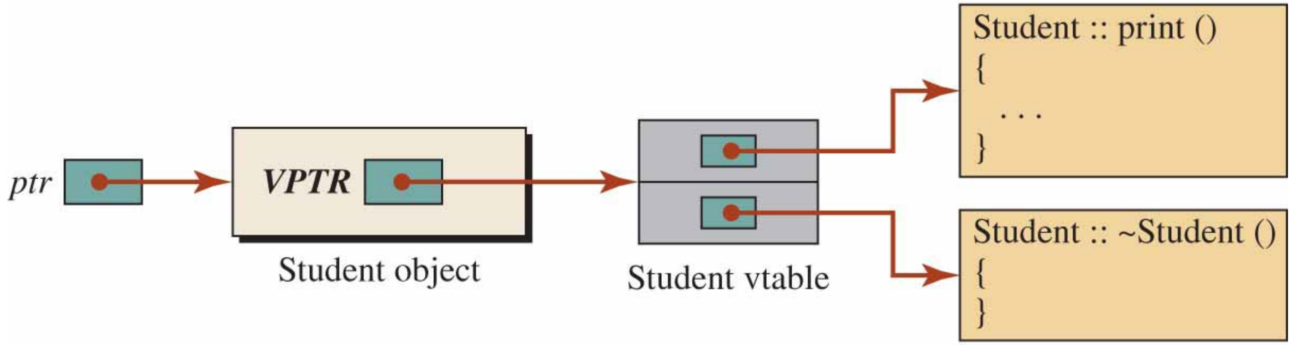
Case 2: Polymorphism Part 2

The Figure shows that we have two entries in the virtual table and when an object in the heap is deleted, the program knows which destructor should be called.

Virtual tables when using virtual destructors



When *ptr* is pointing to the Person class



When *ptr* is pointing to the Student class

In-class Exercise #1

Add constructors and destructors to code example #2, and include *cout* messages in the constructors and destructors for *Base* and *Derived* to show the call order.

Check the call order and resolve any issues to ensure the correct call order is shown.

Code Example #3

C++ recommends that we always define an explicit destructor for the base class in polymorphism and make it virtual.
Use of virtual destructors prevents possible memory leaks in polymorphism.

Example

We show how we can use polymorphism to print the information in the *Student* class through a pointer pointing to the *Person* class.

Interface File for the Person Class Part 1

File person.h

```
/*****************************************************************
 * The interface file for the Person class                      *
 *****************************************************************/
#ifndef PERSON_H
#define PERSON_H
#include <iostream>
#include <string>
using namespace std;

class Person
{
    private:
        string name;
    public:
        Person (string name);
        virtual ~Person ();
        virtual void print () const;
};
#endif
```

Interface File for the Person Class Part 2

File person.cpp

```
/*****************************************************************
 * The implementation file for the Person class                 *
 *****************************************************************/
#include "Person.h"

// Definition of the Person constructor
Person :: Person (string nm)
: name (nm)
{
}
// Definition of the Person destructor (virtual)
Person :: ~Person ()
{
}
// Definition of the print function (virtual)
void Person :: print () const
{
     cout << "Name: " << name << endl;
}
```

Interface File for the Person Class Part 3

File student.h

```
/***********************************************************************
 * The interface file for the Student class                            *
 ***********************************************************************/
#ifndef STUDENT_H
#define STUDENT_H
#include "person.h"

class Student: public Person
{
    private:
        double gpa;
    public:
        Student (string name, double gpa);
        virtual void print () const;
};
#endif
```

Implementation File for the Person Class

File student.cpp

```
/****************************************************************
 * The implementation file for the Student class               *
 ****************************************************************/
#include "Student.h"

// Definition of Constructor for Student class
Student :: Student (string nm, double gp)
: Person (nm), gpa (gp)
{
}
// Definition of virtual print function for Student class
void Student :: print () const
{
    Person :: print ();
    cout << "GPA: " << gpa << endl;
}
```

Application File for the Person Class

```
/*************************************************************
 * The application file to test Person and Student classes     *
 *************************************************************/
#include "Student.h"

int main ( )
{
     // Creation of ptr as polymorphic variable
     Person* ptr;
     // Instantiation Person object in the heap
     ptr = new Person ("Lucie");
     cout << "Person Information";
     ptr -> print();
     cout << endl;
     delete ptr;
     // Instantiation Student object in the heap
     ptr = new Student ("John", 3.9);
     cout << "Student Information";
     ptr -> print();
     cout << endl;
     delete ptr;
     return 0;
}
```

```
Run:
Person Information
Name: Lucie
Student Information
Name: John
GPA: 3.9
```

In-class Exercise #2

Add constructors and destructors to code example #3, and include *cout* messages in the constructors and destructors for *Person* and *Student* to show the call order.

Check the call order and resolve any issues to ensure the correct call order is shown.

Better Use of Polymorphism

The previous programs show the idea of polymorphism with only one pointer that can point to different objects.

A better demonstration is when we have to use polymorphism. Assume we need to have an array of objects.

We know that all elements of an array needs to be of the same type; this means we cannot use an array of objects if the objects are of different types.

However, we can use an array of pointers, in which each pointer can point to an object of the base class (*Person* in the previous example).

In other words, we can have an array of polymorphic variables, instead of one.

Code Example #4: Using an Array of Pointers in Polymorphism Part 1

```
/*****************************************************************
 * Modification of application file to show the actual use of  *
 * polymorphism with an array of poiners.                      *
 *****************************************************************/
#include "student.h"

int main ( )
{
     // Declaration of an array of polymorphic variables (pointers)
     Person* ptr [4];
     // Instantiation of four objects in the heap memory
     ptr[0] = new Person ("Bruce");
     ptr[1] = new Person ("Sue");
     ptr[2] = new Student ("Joe", 3.7);
     ptr[3] = new Student ("John", 3.9);
     // Calling the virtual print function for each object
     for (int i = 0; i < 4; i++)
     {
         ptr[i] -> print ();
         cout << endl;
```

Code Example #4: Using an Array of Pointers in Polymorphism Part 2

```
    }
    // Deleting the objects in the heap
    for (int i = 0; i < 4; i++)
    {
        delete ptr [i];
    }
    return 0;
}
```

```
Run:

Name: Bruce

Name: Sue

Name: Joe
GPA: 3.70

Name: John
GPA: 3.90
```

Object Binding

Object Binding

An issue related to polymorphism that needs to be discussed is *binding*.

We know that a function is split into two entities: function call and function definition.

Binding here means the association between the function call for example, *print* (), and the function body, for example, *void print* {…}.

We know that we may have two functions with the same signature (overriding functions). This means that the function has only one form of call, but may have more than one definition.

If a Person object calls the print function, one definition is executed; if a Student object calls the print function, a different definition is executed.

Binding here means how the program binds (associates) a function call to a function definition. There are two cases: *static binding* and *dynamic binding.*

Static Binding

The term *static binding* (sometimes called *compile-time binding* or *early binding*) occurs when we have more than one definition for a function, but the compiler exactly knows which version of the definition is to be used when the program is compiled.

This may happens, for example, when the function is called by its corresponding object as show below.

```
person.print();
student.print();
```

When the compiler encounters the first call, it knows that the call should be for the definition of print function that prints the data member of the Person object.

When the compiler encounters the second call, it knows that the call should be to the definition of the print function that prints the data members of the Student object.

Dynamic Binding

We use polymorphism for *dynamic binding* (also called *late binding* or *run-time binding*), which means that we need to bind a call to the corresponding definition during run time.

This is needed when the object is not known during the compilation.

For this reason, we need a virtual function to force the run-time system to create a table that shows which object needs which function, and bind the call to the appropriate function.

Polymorphism is closely tied to dynamic binding because we want to be able to execute the appropriate function definition.

Run-Time Type Information (RTTI)

When working with hierarchy of classes, sometimes we need to know the type of the object we are dealing with or sometimes we want to change the type of the object.

Using typeid Operator

To determine the type of the object at run time, we can use the <typeinfo> header to access an object of the class *type_info*.
It has no constructor, destructor, or copy constructor.
We create an object of *type_info* using an overloaded operator named *typeid*.
We can create an object of *type_info* by passing an expression to the operator *typeid* that can be evaluated as a type

For example *typeid* (5), *typeid* (object_name), *typeid* (6 + 2), and so on.

We can then uses one of the four member functions or operators that are defined in the *type_info* class as shown in the Table below in which *t1* and *t2* are object of *type_info* class.

Operation on type_info objects

```
t1 == t2          // Returns true if t1 and t2 are of the same type
t1 != t2          // Returns true if t1 and t2 are of different types
t1.name()         // Returns a C-type string (name of the t1)
t1.before(t2)     // Returns true if t1 comes before t2
```

Code Example #5: Testing typeid Operator

```
/*****************************************************************
 * A program to use typeid operator to find the name of classes *
 *****************************************************************/
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

class Animal {};
class Horse {};

int main ( )
{
    Animal a;
    Horse h;
    cout << typeid(a).name() << endl;
    cout << typeid(h).name();
    return 0;
}
```

Run:

```
6Animal
5Horse
```

Or:

```
class Animal
class Horse
```

The name of the class is preceded by the number of character in each case (6 and 5).

Using Dynamic-Cast Operator

We have seen that in a polymorphic relationship we can *upcast* a pointer, which means to make a pointer to a derived class to be assigned to a pointer to a base class as shown below:

```
Person*  ptr1 = new Student
```

Here the pointer returned from the *new* operator is a pointer to a *Student* object, but we assign it to a pointer that points to a *Person* object (the pointer is upcast).

C++ also allows us to downcast a pointer to make it to point to an object in the lower order of hierarchy.

This can be done using a *dynamic_cast* operator as shown below (*ptr*1 is a pointer to *Person* object).

```
Student*  ptr2 = dynamic_cast <Student*)(ptr1);
```

This casting proves that the *Student* class is a class derived from the *Person* class because *ptr1* can be downcast to *ptr2*.

However it is not recommended because of the overhead involved.

Abstract Classes

Abstract Classes Part 1

When we design a set of classes, sometimes we find that there is a list of behaviors that are identical to all classes.

For example, assume we define two classes named *Rectangle* and *Square*.

Both of these classes have at least two common behaviors: *getArea*() and *getPerimeter*().

How can we force the creator of these two classes to provide the definition of both member functions for each class? (as a requirement)

We know that the formula to find the area and perimeter of these geometrical shapes is different; which means that each class must have its own version of *getArea* () and *getPerimeter* ().

The solution in object-oriented programming is to declare an *abstract* class, that forces the creators of all derived classes to remember to add these two definitions to their classes.

A set of classes with one abstract class must have the declaration and definition for the *pure virtual functions*.

An *abstract class* is a class with at least one *pure virtual function*.

Abstract Classes Part 2

Declaration of Pure Virtual Functions

An *abstract class* is a class with at least one *pure virtual function*.

A *pure virtual function* is a virtual function in which the declaration is set to zero and which has no definition in the *abstract class*.

The following shows two *virtual member function* for the *Shape* class:

```
virtual double getArea(0) = 0;
virtual double getPerimeter(0) = 0;
```

Definition of Pure Virtual Functions

The abstract class does not define its pure virtual function, but every class that inherits from the abstract class needs to provide the definition of each pure virtual function or declared it as a pure virtual to be defined in the next lower level of the hierarchy.

No Instantiation

We cannot instantiate an object from an abstract class because it does not have the definition of its pure virtual functions.

For an object to be able to instantiated from a class, the class needs to have the definition of all member function.

This means that an abstract class needs to be polymorphically inherited to define concrete classes for instantiation.

An abstract class cannot be instantiated because there is no definition for the pure virtual member functions.

Interfaces

An abstract class can have both virtual and pure virtual functions.

In some cases, however, we may need to create a blue print for inherited classes.

We can define a class with all pure virtual functions.

This class is sometimes referred to as an interface; we cannot create any implementation file from this class, only the interface file.

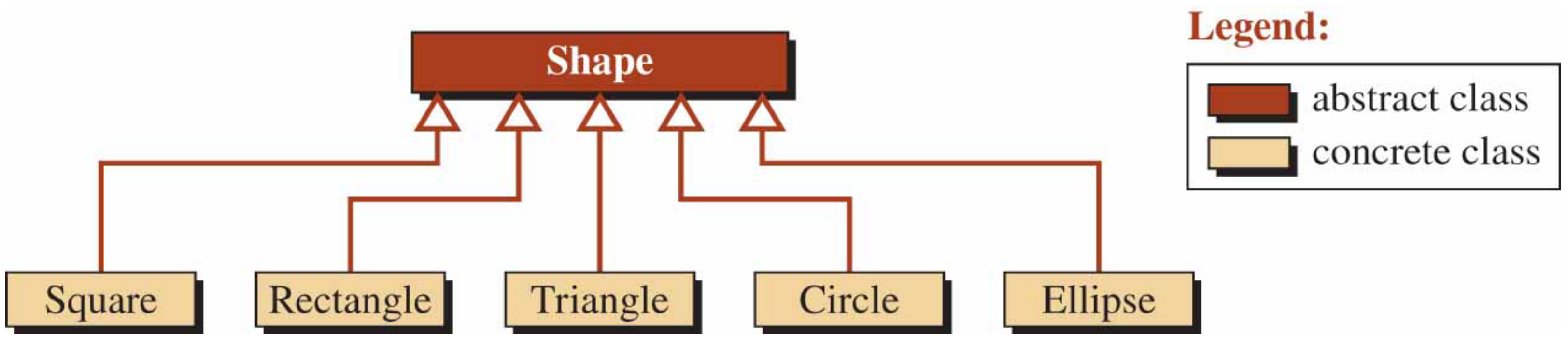
An interface is a special case of an abstract class in which all member functions are pure virtual.

Code Example #6: Interfaces

Example

We create five concrete classes to represent shapes. All classes are inherited from an abstract class Shape.

Adding an abstract class to a set of classes



Interface for the Abstract Shape Class

File shape.h

```
/*****************************************************************
 * The interface for the abstract Shape class                    *
 *****************************************************************/
#ifndef SHAPE_H
#define SHAPE_H
#include <iostream>
#include <cassert>
#include <cmath>
using namespace std;

// Class definition
class Shape
{
    protected:
        virtual bool isValid () const = 0;
    public:
        virtual void print () const = 0 ;
        virtual double getArea () const = 0 ;
        virtual double getPerimeter () const = 0;
};
#endif
```

Interface File of the Square Class

File square.h

```
/*****************************************************************
 * The interface file the Square class                          *
 *****************************************************************/
#ifndef SQUARE_H
#define SQUARE_H
#include "shape.h"

// Class Definition
class Square : public Shape
{
    private:
        double side;
        bool isValid() const;
    public:
        Square (double side);
        ~Square ();
        void print() const;
        double getArea () const;
        double getPerimeter () const;
};
#endif
```

Implementation File of the Square Class Part 1

File square.cpp

```
/*****************************************************************
 * The implementation file the Square class                      *
 *****************************************************************/
#include "square.h"

// Constructor
Square :: Square (double s)
:side (s)
{
     if (!isValid ())
     {
         cout << "Invalid square!";
         assert (false);
     }
}
// Destructor
Square :: ~Square ()
{
}
// Definition of print function
```

Implementation File of the Square Class Part 2

File square.cpp

```
void Square :: print () const
{
     cout << "Square of side " << side << endl;
}
// Finding the area
double Square :: getArea () const
{
     return (side * side);
}
// Finding the perimeter
double Square :: getPerimeter () const
{
     return (4 * side);
}
// Private isValid function
bool Square :: isValid () const
{
     return (side > 0.0);
}
```

Interface File for the Rectangle Class

File rectangle.h

```
/***************************************************************
 * The interface file for the Rectangle class                  *
 ***************************************************************/
#ifndef RECTANGLE_H
#define RECTANGLE_H
#include "shape.h"

// Class definition
class Rectangle : public Shape
{
    private:
        double length;
        double width;
        bool isValid() const;
    public:
        Rectangle (double length, double width);
        ~Rectangle ();
        void print () const;
        double getArea() const;
        double getPerimeter() const;
}
#endif
```

Implementation File the Rectangle Class Part 1

File rectangle.cpp

```
/**************************************************************
 * The implementation file the Rectangle class               *
 **************************************************************/
#include "rectangle.h"

// Constructor
Rectangle :: Rectangle (double lg, double wd)
: length (lg), width (wd)
{
    if (!isValid())
    {
        cout << "Invalid rectangle!";
        assert (false);
    }
}
// Destructor
Rectangle :: ~Rectangle ()
{
}
// Definition of print function
```

Implementation File the Rectangle Class Part 2

File rectangle.cpp

```
void Rectangle :: print () const
{
     cout << "Rectangle of " << length << " X " << width << endl;
}
// Finding the area
double Rectangle :: getArea() const
{
     return length * width;
}
// Finding the perimeter
double Rectangle :: getPerimeter() const
{
     return 2 * (length + width);
}
// Private isValid function
bool Rectangle :: isValid () const
{
     return (length > 0.0 && width > 0.0);
}
```

Interface File for the Triangle Class

File triangle.h

```cpp
/***********************************************************************
 * The interface file for the Triangle class                          *
 ***********************************************************************/
#ifndef TRIANGLE_H
#define TRIANGLE_H
#include "shape.h"

// Class definition
class Triangle : public Shape
{
  private:
        double side1;
        double side2;
        double side3;
        bool isValid () const;
  public:
        Triangle (double side1, double side2, double side3);
        ~Triangle ();
        void print() const;
        double getArea() const;
        double getPerimeter() const;
}
#endif
```

Implementation File the Triangle Class Part 1

File triangle.cpp

```
/***************************************************************
 * The implementation file the Triangle class                  *
 ***************************************************************/
#include "triangle.h"

// Constructor
Triangle :: Triangle (double s1, double s2, double s3)
: side1(s1), side2(s2), side3 (s3)
{
    if (!isValid())
    {
        cout << "Invalid triangle!";
        assert (false);
    }
}
// Destructor
Triangle :: ~Triangle ()
{
}
// Definition of print function
```

Implementation File the Triangle Class Part 2

```
void Triangle :: print() const
{
     cout << "Triangle of : " << side1 << " X " << side2 << " X ";
     cout << side3 << endl;
}
// Finding the area
double Triangle :: getArea() const
{
     double s = (side1 + side2 + side3) / 2;
     return (sqrt (s * (s - side1) * (s - side2) * (s - side3)));
}
// Finding the perimeter
double Triangle :: getPerimeter() const
{
     return (side1 + side2 + side3);
}
// Private isValid function
bool Triangle :: isValid () const
{
     bool fact1 = (side1 + side2) > side3;
     bool fact2 = (side1 + side3) > side2;
     bool fact3 = (side2 + side3) > side1;
     return (fact1 && fact2 && fact3);
}
```

Interface File for the Circle Class

File circle.h

```
/****************************************************************
 * The interface file for the Circle class                     *
 ****************************************************************/
#ifndef CIRCLE_H
#define CIRCLE_H
#include "shape.h"

// Class definition
class Circle : public Shape
{
    private:
        double radius;
        bool isValid () const;
    public:
        Circle (double radius);
        ~Circle ();
        void print() const;
        double getArea() const;
        double getPerimeter() const;
};
#endif
```

Implementation File the Circle Class Part 1

File circle.cpp

```
/******************************************************************
 * The implementation file the Circle class                       *
 ******************************************************************/
#include "circle.h"

// Constructor
Circle :: Circle (double r)
: radius (r)
{
     if (!isValid())
     {
         cout << "Invalid circle!";
         assert (false);
     }
}
// Destructor
Circle :: ~Circle ()
{
}
// Definition of print function
```

Implementation File the Circle Class Part 2

File circle.cpp

```
void Circle :: print() const
{
     cout << "Circle of radius : " << radius << endl;
}
// Finding the area
double Circle :: getArea() const
{
     return (3.14 * radius * radius);
}
// Finding the perimeter
double Circle :: getPerimeter() const
{
     return 2 * 3.14 * radius;
}
// Private isValid function
bool Circle :: isValid () const
{
     return (radius > 0);
}
```

Interface File for the Ellipse Class

File ellipse.h

```
/*****************************************************************
 * The interface file for the Ellipse class                      *
 *****************************************************************/
#ifndef ELLIPSE_H
#define ELLIPSE_H
#include "shape.h"

// Class definition
class Ellipse : public Shape
{
    private:
        double radius1;
        double radius2;
        bool isValid () const;
    public:
        Ellipse (double radius1, double radius2);
        ~Ellipse ();
        void print() const;
        double getArea () const;
        double getPerimeter () const;
}
#endif
```

Implementation File for the Ellipse Class Part 1

File ellipse.cpp

```
/*****************************************************************
 * The interface file for the Ellipse class                      *
 *****************************************************************/
#include "ellipse.h"

// Constructor
Ellipse :: Ellipse (double r1, double r2)
: radius1 (r1), radius2 (r2)
{
     if (!isValid())
     {
         cout << "Invalid ellipse!";
         assert (false);
     }
}
// Destructor
Ellipse :: ~Ellipse ()
{
}
// Definition of print function
```

Implementation File for the Ellipse Class Part 2

File ellipse.cpp

```
void Ellipse :: print() const
{
     cout << "Ellipse of radii: " << radius1 << " X " <<;
     cout << radius2 << endl;
}
// Finding the area
double Ellipse :: getArea () const
{
     return (3.14 * radius1 * radius2);
}
// Finding the perimeter
double Ellipse ::getPerimeter () const
{
     double temp = (radius1 * radius1 + radius2 * radius2) / 2;
     return (2 * 3.14 * temp);
}
// Private isValid function
bool Ellipse :: isValid () const
{
     return (radius1 > 0 && radius2 > 0);
}
```

Application File to Test All Classes Part 1

```
/***************************************************************
 * The application file to test all classes                   *
 ***************************************************************/
#include "square.h"
#include "rectangle.h"
#include "triangle.h"
#include "circle.h"
#include "ellipse.h"

int main ( )
{
    // Instantiation and testing the Square class
    cout << "Information about a square" << endl;
    Square square (5);
    square.print ();
    cout << "area: " << square.getArea () << endl;
    cout << "Perimeter: " << square.getPerimeter () << endl;
    cout << endl;
    // Instantiation and testing the Rectangle class
    cout << "Information about a rectangle" << endl;
    Rectangle rectangle (5, 4);
    rectangle.print ();
    cout << "area: " << rectangle.getArea () << endl;
    cout << "Perimeter: " << rectangle.getPerimeter () << endl;
    cout << endl;
```

Application File to Test All Classes Part 2

File app.cpp

```
        // Instantiation and testing the Triangle class
        cout << "Information about a triangle" << endl;
        Triangle triangle (3, 4, 5);
        triangle.print ();
        cout << "area: " << triangle.getArea () << endl;
        cout << "Perimeter: " << triangle.getPerimeter () << endl;
        cout << endl;
        // Instantiation and testing the Circle class
        cout << "Information about a circle" << endl;
        Circle circle (5);
        circle.print ();
        cout << "area: " << circle.getArea () << endl;
        cout << "Perimeter: " << circle.getPerimeter () << endl;
        cout << endl;
        // Instantiation and testing the Ellipse class
        cout << "Information about an ellipse" << endl;
        Ellipse ellipse (5, 4);
        ellipse.print ();
        cout << "area: " << ellipse.getArea () << endl;
        cout << "Perimeter: " << ellipse.getPerimeter ()<< endl;
        return 0;
}
```

Application File to Test All Classes Part 3

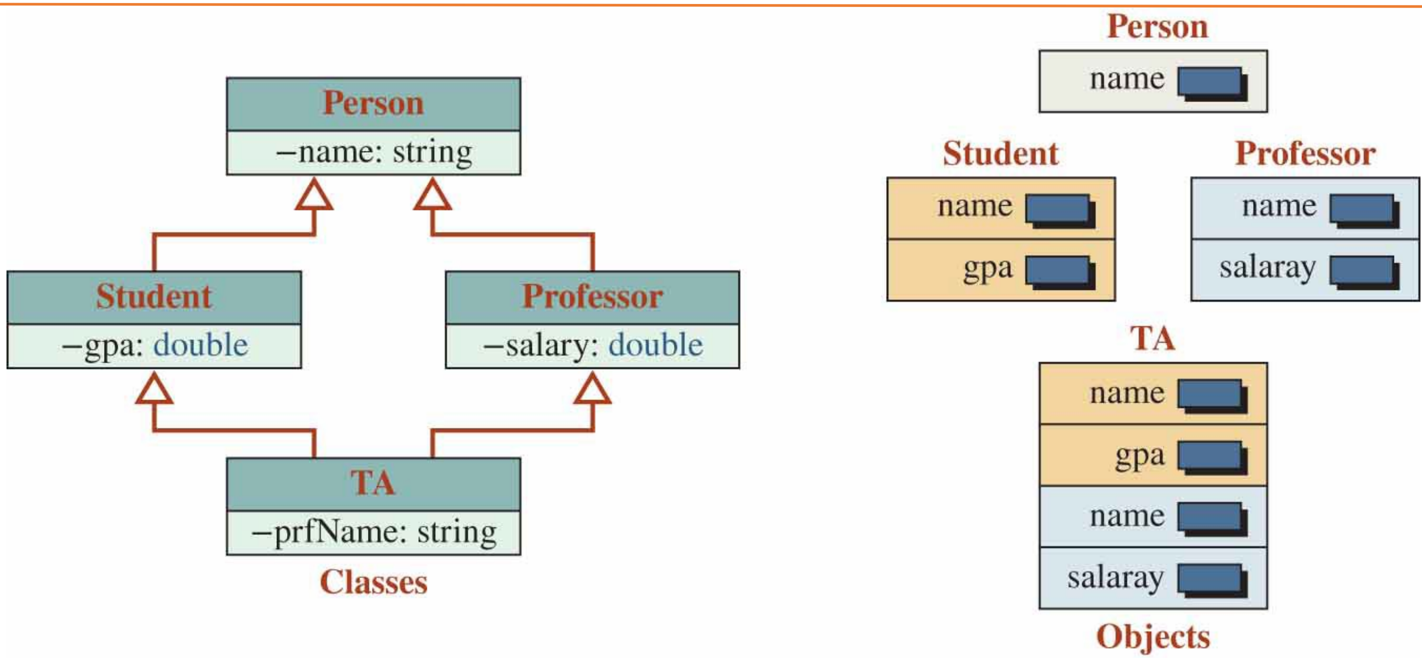
```
Run:
Information about a square
Square of size 5
area: 25
Perimeter: 20

Information about a rectangle
Rectangle of 5 X 4
area: 20
Perimeter: 18

Information about a triangle
Triangle of : 3 X 4 X 5
area: 6
Perimeter: 12


Information about a circle
Circle of radius : 5
area: 78.5
Perimeter: 31.4


Information about an ellipse
Ellipse of radii: 5 X 4
area: 62.8
Perimeter 28.4339
```

Multiple Inheritance

Multiple Inheritance

C++ allows us to derive a class from more than one class. As a simple example, we can have a class named TA (teaching assistant) that is inherited from two classes: Student and Professor as shown in the Figure.

Classes and objects in multiple inheritance



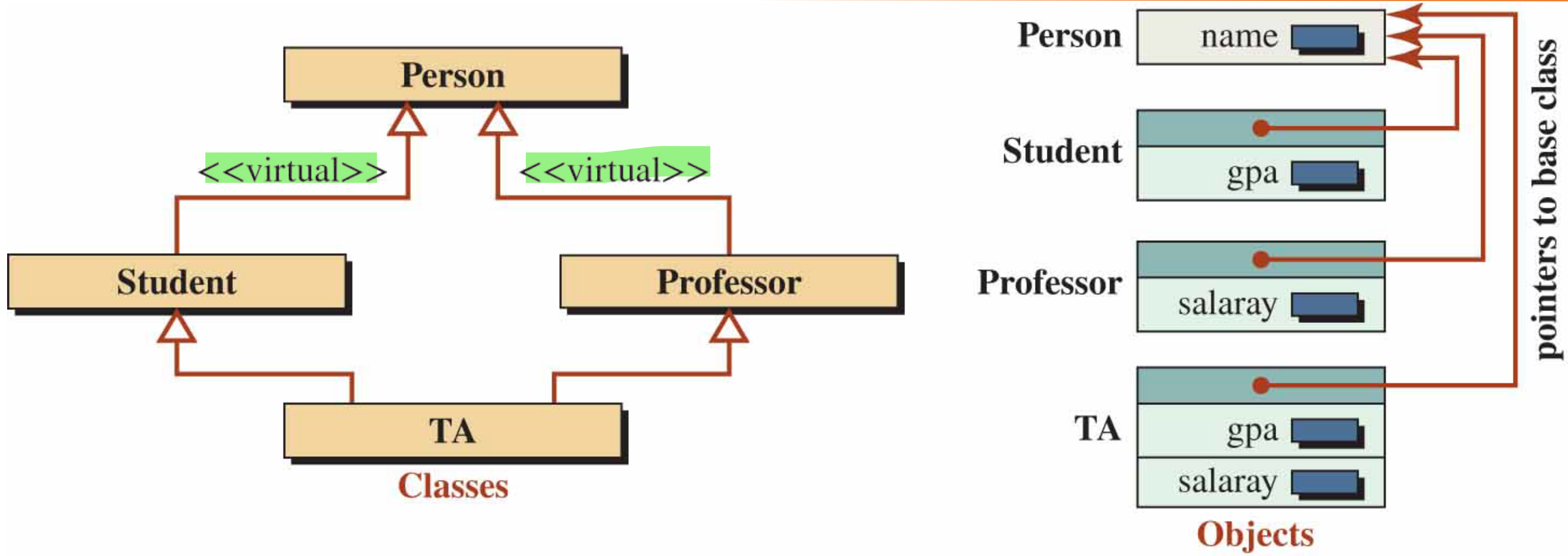
Unfortunately, the inheritance fails when we code these classes because the TA class inherits the data member name from both Student and Professor class.

Virtual Base in Inheritance Part 1

One solution for the problem of duplicated shared data members in multiple inheritance is to use *virtual base inheritance*.

In this type of inheritance, two classes can inherit from a common base using the *virtual* keyword.

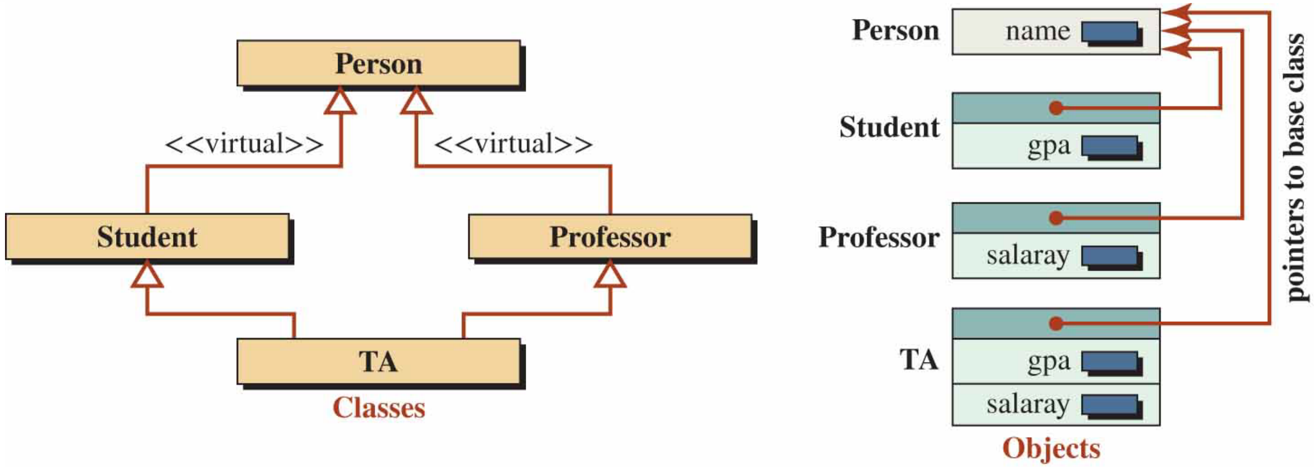
Classes and objects in virtual base inheritance



In this case, we have the following four classes.

```
class Person {…};
class Student: virtual public Person {…};
class Professor: virtual public Person {…};
class TA: public Student, public Professor {…};
```

Virtual Base in Inheritance Part 2

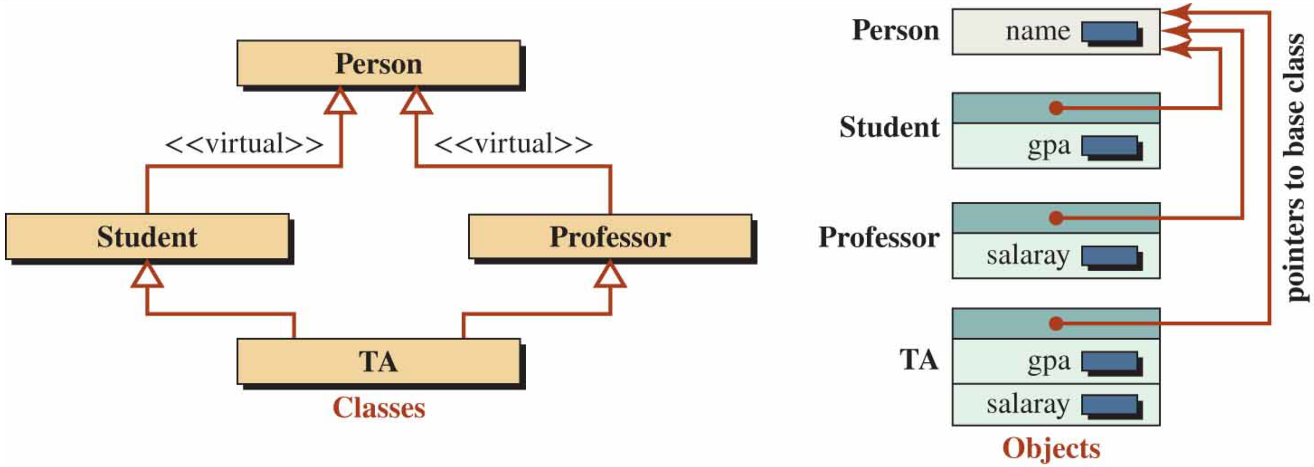



```
class Person {…};
class Student: virtual public Person {…};
class Professor: virtual public Person {…};
class TA: public Student, public Professor {…};
```

When we use virtual base inheritance, the object of the virtual base class is not stored in each object of the derived class.

The object of the virtual base class is stored separately and each derived class has a pointer to this object.

Virtual Base in Inheritance Part 3



One problem when using the virtual base technique is that we need to avoid *delegation* as discussed in the previous chapter.

In other words, we cannot define a print function in the TA class by calling the corresponding print functions in the Student and Professor class because the data member name would be printed three times.

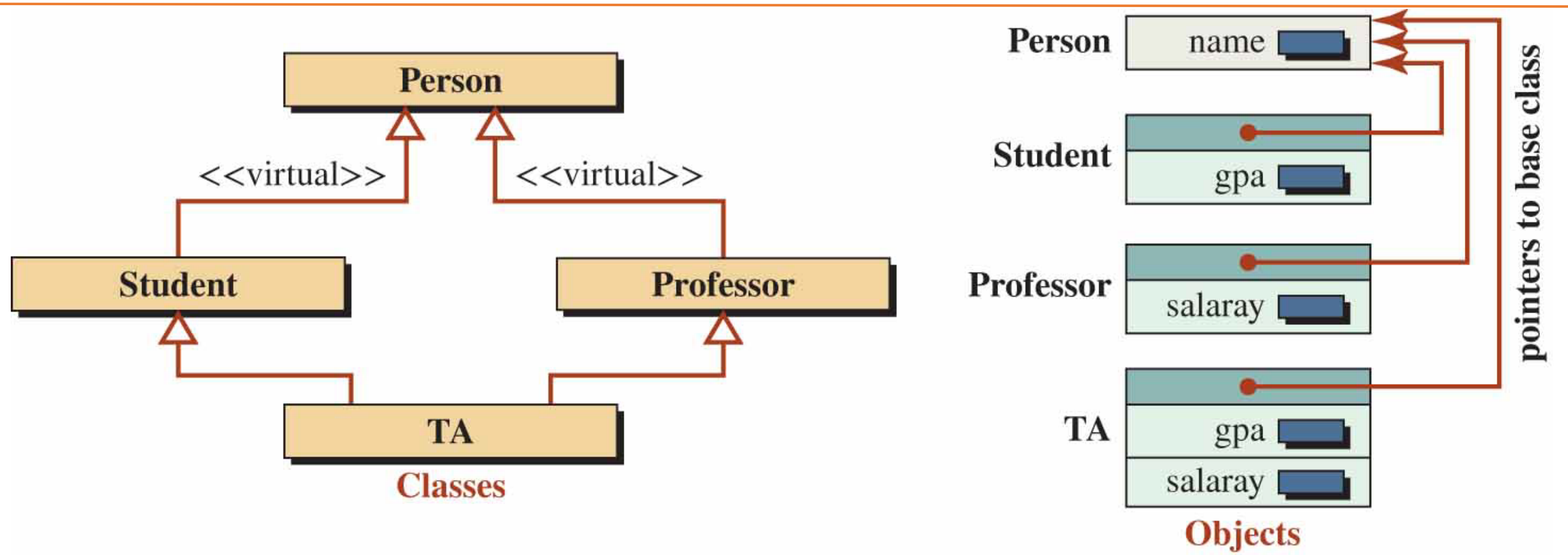
There is more than a solution to this dilemma; the one that we recommend is to make the common data members protected to be seen in all derived classes and avoid using delegated member functions.

Code Example #7: Virtual Base in Inheritance

Example

Try this example with and without the 'virtual' keyword.

Classes and objects in virtual base inheritance



Interface File for the Person Class

File person.h

```
/***************************************************************
 * The interface file for the Person class                    *
 ***************************************************************/
#ifndef PERSON_H
#define PERSON_H
#include <iostream>
#include <cassert>
using namespace std;

class Person
{
    protected:
        string name; // Protected data member
    public:
        Person (string name);
        ~Person ();
        void print ();
};
#endif
```

Implementation File for the Person Class

File person.cpp

```
/*******************************************************************
 * The implementation file for the Person class                 *
 *******************************************************************/
#include "person.h"

// Constructor
Person :: Person (string nm)
: name (nm)
{
}
// Destructor
Person :: ~Person ()
{
}
// Print member function
void Person :: print ()
{
    cout << "Person" << endl;
    cout << "Name: " << name << endl << endl;
}
```

Interface File for the Student Class

File student.h

```
/****************************************************************
 * The interface file for the Student class                     *
 ****************************************************************/
#ifndef STUDENT_H
#define STUDENT_H
#include "person.h"

class Student: virtual public Person // Virtual inheritance
{
    protected:
        double gpa; // Protected data member
    public:
        Student (string name, double gpa);
        ~Student ();
        void print ();
};
#endif
```

Implementation File for the Student Class

File student.cpp

```
/*****************************************************************
 * The implementation file for the Student class                 *
 *****************************************************************/
#include "Student.h"

// Constructor
Student :: Student (string name, double gp)
: Person (name), gpa (gp)
{
     assert (gpa <= 4.0);
}
// Destructor
Student :: ~Student()
{
}
// Print member function uses a protected data member (name)
void Student :: print ()
{
     cout << "Student " << endl;
     cout << "Name: " << name << " ";
     cout << "GPA: " << gpa << endl << endl;
}
```

Interface File for the Professor Class

File professor.h

```
/***********************************************************************
 * The interface file for the Professor class                          *
 ***********************************************************************/
#ifndef PROFESSOR_H
#define PROFESSOR_H
#include "person.h"

class Professor: virtual public Person // Virtual inheritance
{
    protected:
        double salary; // Protected data member
    public:
        Professor (string name, double salary);
        ~Professor ();
        void print ();
};
#endif
```

Implementation File for the Professor Class

File professor.cpp

```
/***************************************************************
 * The implementation file for the Professor class             *
 ***************************************************************/
#include "professor.h"

// Constructor
Professor :: Professor (string nm, double sal)
: Person (nm), salary (sal)
{
}
// Destructor
Professor :: ~Professor ()
{
}
// Print member function
void Professor :: print ()
{
    cout << "Professor " << endl;
    cout << "Name: " << name << " ";
    cout << "Salary: " << salary << endl << endl;
}
```

Interface File for TA Class

File ta.h

```
/*****************************************************************
 * The interface file for the TA class                           *
 *****************************************************************/
#ifndef TA_H
#define TA_H
#include "student.h"
#include "professor.h"

class TA: public Professor, public Student // Double inheritance
{
    public:
        TA (string name, double gpa, double sal);
        ~TA ();
    void print ();
};
#endif
```

Implementation File for the TA Class

File ta.cpp

```
/***************************************************************
 * The implementation file for the TA class                   *
 **************************************************************/
#include "ta.h"

// Constructor
TA :: TA (string nm, double gp, double sal)
: Person (nm), Student (nm, gp), Professor (nm, sal)
{
}
// Destructor
TA :: ~TA ()
{
}
// Print member function
void TA :: print ()
{
    cout << "Teaching Assistance: " << endl;
    cout << "Name: " << name << " ";
    cout << "GPA: " << gpa << " ";
    cout << “Salary: “ << salary << endl << endl;
}
```

Application to Test All Four Classes Part 1

File application.cpp

```
/***************************************************************
 * The application to test all four classes (Person, Student,  *
 * Professor, and TA).                                         *
 ***************************************************************/
#include "ta.h"

int main ( )
{
    // Testing Person class
    Person person ("John");
    person.print ();
    // Testing Student class
    Student student ("Anne", 3.9);
    student.print ();
    // Testing Professor class
    Professor professor ("Lucie", 78000);
    professor.print ();
    // Testing TA class
    TA ta ("George", 3.2, 20000);
    ta.print ();
    return 0;
}
```

Application to Test All Four Classes Part 2

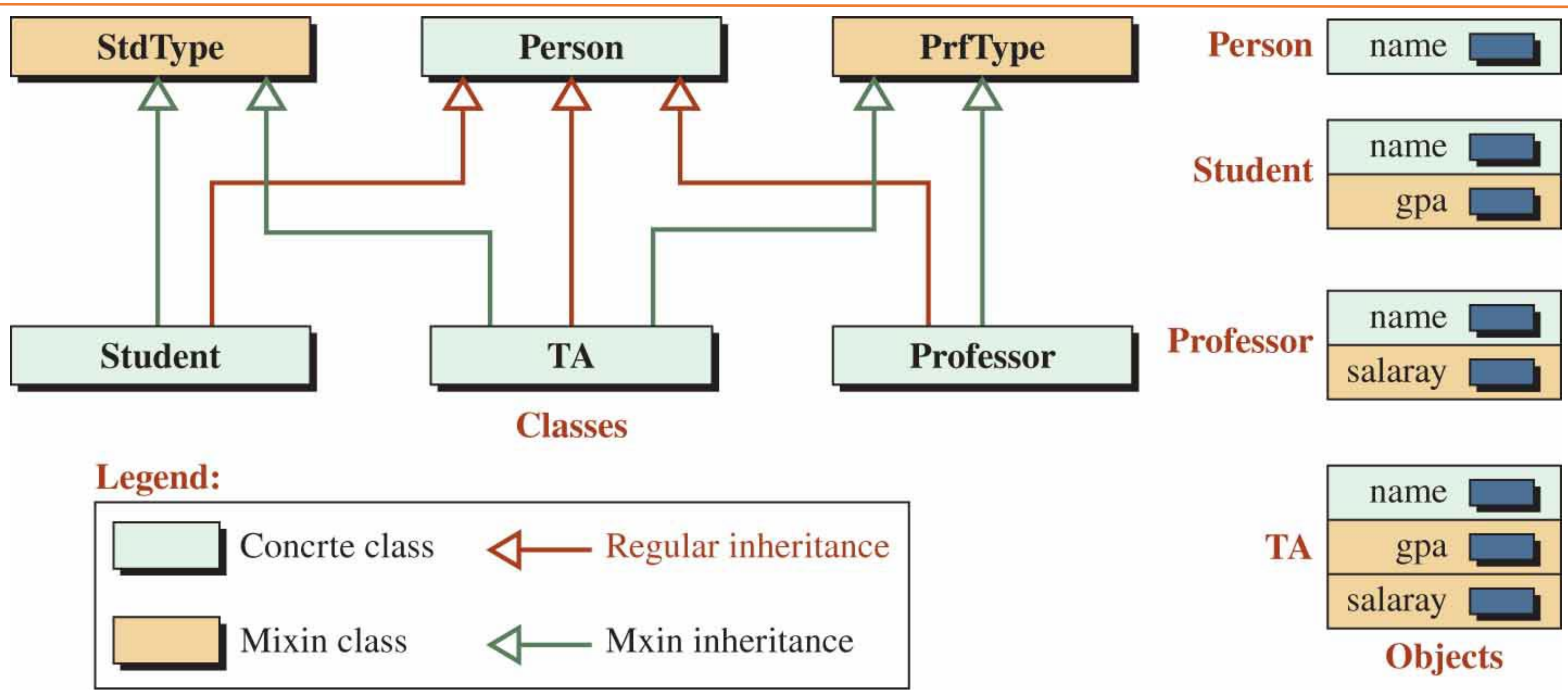
```
Run:
Person
Name: John

Student
Name: Anne GPA: 3.9

Professor
Name: Lucie Salary: 78000

Teaching Assistance:
Name: George GPA: 3.2 Salary: 20000
```

Multiple Inheritance Using Mixin Classes

Another solution for the problem of common base classes in multiple inheritance is the use of mixin classes.

A mixin class is never instantiated (it has some pure virtual functions), but it can add data members to other classes.

Multiple inheritance using mixin classes



Interface File for StdType Abstract Class

File stdtype.h

```
/*****************************************************************
 * The interface file for StdType abstract class               *
 *****************************************************************/
#ifndef STDTYPE_H
#define STDTYPE_H
#include <iostream>
using namespace std;

class StdType
{
    protected:
        double gpa;
    public:
        virtual void printGPA( ) = 0 ;
};
#endif
```

Interface File for PrfType Abstract Class

File prftype.h

```
/****************************************************************
 * The interface file for PrfType abstract class               *
 ****************************************************************/
#ifndef PRFTYPE_H
#define PRFTYPE_H
#include <iostream>
using namespace std;

class PrfType
{
    protected:
        double salary;
    public:
        virtual void printSalary () = 0;
};
#endif
```

Interface File for Person Concrete Class

File person.h

```
/*****************************************************************
 * The interface file for Person concrete class                 *
 *****************************************************************/
#ifndef PERSON_H
#define PERSON_H
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;

class Person
{
    private:
        string name;
    public:
        Person (string name);
        void print ();
};
#endif
```

Interface File for Student Concrete Class

File student.h

```
/*****************************************************************
 * The interface file for Student concrete class. This class    *
 * inherits from two classes: Person and StdType.               *
 *****************************************************************/
#ifndef STUDENT_H
#define STUDENT_H
#include "person.h"
#include "stdtype.h"

class Student: public Person, public StdType
{
    public:
        Student (string name, double gpa);
        void printGPA();
        void print();
};
#endif
```

Interface File for Professor Concrete Class

File professor.h

```cpp
/***************************************************************
 * The interface file for Professor concrete class. This class *
 * inherits from two classes: Person and PrfType.              *
 ***************************************************************/
#ifndef PROFESSOR_H
#define PROFESSOR_H
#include "person.h"
#include "prftype.h"

class Professor : public Person, public PrfType
{
    public:
        Professor (string name, double salary);
        void printSalary();
        void print ();
};
#endif
```

Interface File for TA Concrete Class

File ta.h

```
/******************************************************************
 * The interface file for TA concrete class. This class          *
 * inherits from tree classes: Person and StdType and PrfType.  *
 ******************************************************************/
#ifndef TA_H
#define TA_H
#include "person.h"
#include "stdtype.h"
#include "prftype.h"

class TA: public Person, public StdType, public PrfType
{
    public:
        TA (string name, double gpa, double salary);
        void printGPA ();
        void printSalary();
        void print ();
};
#endif
```

Implementation File for Person Concrete Class

File person.cpp

```
/***************************************************************
 * The implementation file for Person concrete class          *
 ***************************************************************/
#include "person.h"

// Constructor
Person :: Person (string nm)
: name(nm)
{
}
// Print member function
void Person :: print ( )
{
    cout << "Name: " << name << endl;
}
```

Implementation file for Student Concrete Class

File student.cpp

```
/*****************************************************************
 * The implementation file for Student concrete class           *
 *****************************************************************/
#include "student.h"

// Constructor
Student :: Student (string na, double gp)
:Person (na)
{
    gpa = gp; // Assignment, not initialization
}
// PrintGPA member function
void Student :: printGPA ()
{
    cout << "GPA: " << fixed << setprecision (2) << gpa << endl;
}
// Print member function
void Student :: print ()
{
    Person :: print();
    printGPA ();
}
```

Implementation File for Professor Concrete Class

File professor.cpp

```
/***************************************************************
 * The implementation file for Professor concrete class        *
 ***************************************************************/
#include "professor.h"

// Constructor
Professor :: Professor (string nm, double sal)
: Person (nm)
{
     salary = sal; // Assignment, not initialization
}
// PrintSalary member function
void Professor :: printSalary ()
{
     cout << "Salary: ";
     cout << fixed << setprecision (2) << salary << endl;
}
// General print function
void Professor :: print ()
{
     Person :: print();
     printSalary();
}
```

Implementation File for TA Concrete Class Part 1

File ta.cpp

```cpp
/************************************************************
 * The implementation file for TA concrete class            *
 ************************************************************/
#include "ta.h"

// Constructor
TA :: TA (string nm, double gp, double sal)
: Person (nm)
{
    gpa = gp; // Assignment, not initialization
    salary = sal; // Assignment, not initialization
}
// member function to print GPA
void TA :: printGPA ()
{
    cout << "GPA: " << gpa << endl;
}
// member function to print salary
void TA :: printSalary ()
{
```

Implementation File for TA Concrete Class Part 2

File ta.cpp

```
    cout << "Salary: ";
    cout << fixed << setprecision (2) << salary << endl;
}
// General print function
void TA :: print ()
{
    Person :: print();
    printGPA ();
    printSalary();
}
```

Application File to Test the Three Classes Part 1

File application.cpp

```
/****************************************************************
 * The application file to test the three classes              *
 ****************************************************************/
#include "student.h"
#include "professor.h"
#include "ta.h"

int main ( )
{
    // Instantiation of four objects
    Person per ("John");
    Student std ("Linda", 3.9);
    Professor prf("George", 89000);
    TA ta ("Lucien", 3.8, 23000);
    // Printing information about a person
    cout << "Information about person" << endl;
    per.print();
    cout << endl << endl;
    // Printing information about a student
    cout << "Information about student" << endl;
```

Application File to Test the Three Classes Part 2

File application.cpp

```
    std.print ();
    cout << endl << endl;
    // Printing information about a professor
    cout << "Information about professor" << endl;
    prf.print();
    cout << endl << endl;
    // Printing information about a teaching assistant
    cout << "Information about teaching assistance " << endl;
    ta.print();
    cout << endl << endl;
    return 0;
}
```

Application File to Test the Three Classes Part 3

```
Run:
Information about person
Name: John

Information about student
Name: Linda
GPA: 3.90

Information about professor
Name: George
Salary: 89000.00

Information about teaching assistance
Name: Lucien
GPA: 3.80
Salary: 23000.00
```

Summary

Polymorphic variables.

Virtual functions.

Dynamic bindings.

Abstract Classes. Interfaces.

Multiple Inheritance. Virtual Base Inheritance.

What's Next?

Reading Assignment

- **Read Chap. 13. Operator Overloading**

End of Class

Thank you

E-mail: youngcha@konkuk.ac.kr