Object-oriented Programming
Inheritance

YoungWoon Cha

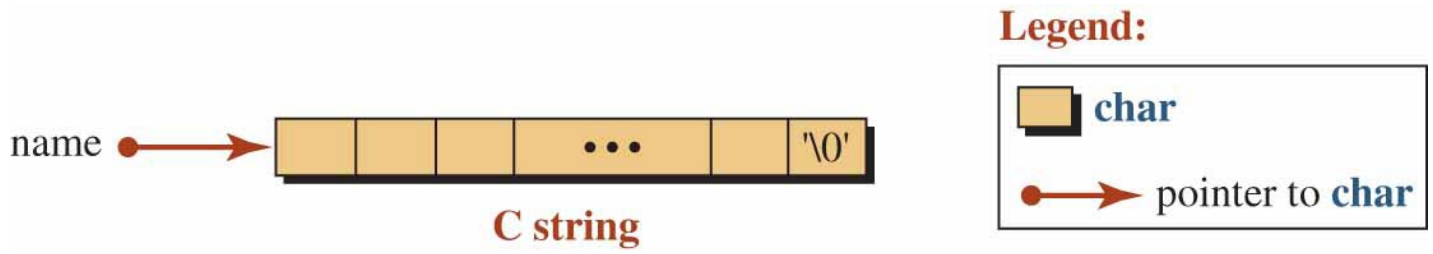
Computer Science and Engineering

KU KONKUK UNIVERSITY

Review

C-STRINGS

Figure 10.1 *General idea behind a C- string*



Since the name of an array is a pointer to the first element in the array, the name of a C-string is a pointer to the first character in the string.
However, we must remember that the name of a C-string does not define a variable; it defines a constant pointer.

The C-string name is a constant pointer to the first character.

C++ string Class

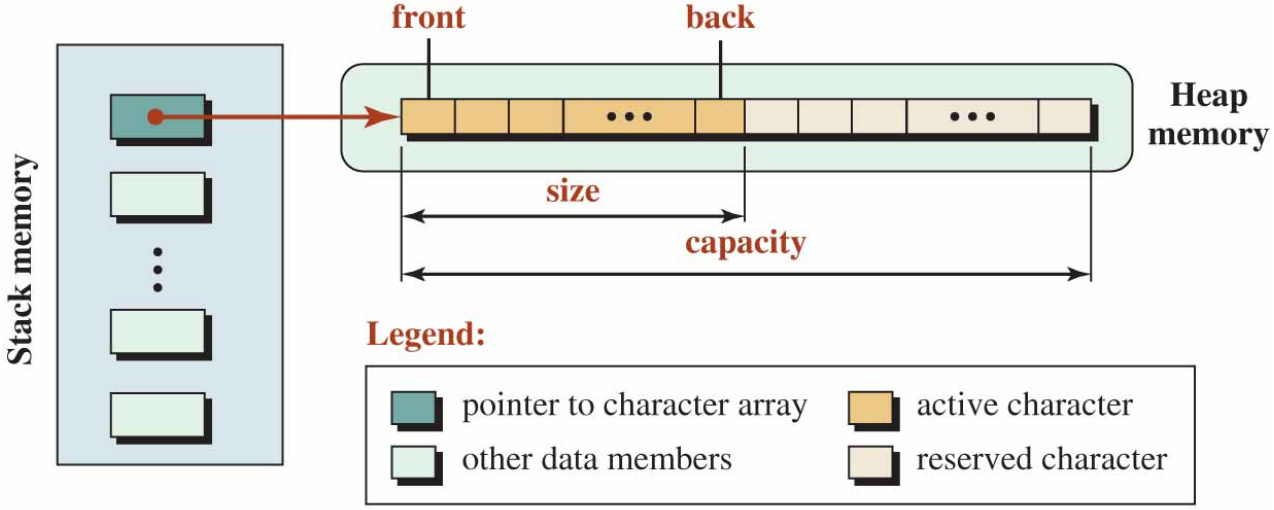
The string class has private data members and public member functions.

The user calls the public member functions to manipulate string objects.

In general, the data members include a pointer to an array of characters.

Other data members keep information about the character array as we discuss gradually.

The data members are normally created in stack memory, but the character array itself is allocated in the heap because its size is not defined until run time.



Notes:

1. A C++ string is not null terminated.
2. Size must be less than or equal to capacity.
3. The front is at index 0.
4. The back is at index (size −1).

A C++ string is an array of characters, but it is not null-terminated.

Conversion in Positional Number System

In computer science, we use different positional numbering system: *binary*, *octal*, *decimal*, and *hexadecimal*.

Each positional numbering system uses a set of symbols and a *base*.

The base defines the total number of symbols used in the system.

Table 10.3 shows the base and the symbols we work with in programming.

Note that the value of symbols A, B, C, D, E, F are 10, 11, 12, 13, 14, and 15 respectively.

Table 10.3 ***Positional number system***

System	Base	Symbols
binary	2	0, 1
octal	8	0, 1, 2, 3, 4, 5, 6, 7
decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

KU KONKUK UNIVERSITY

Inheritance

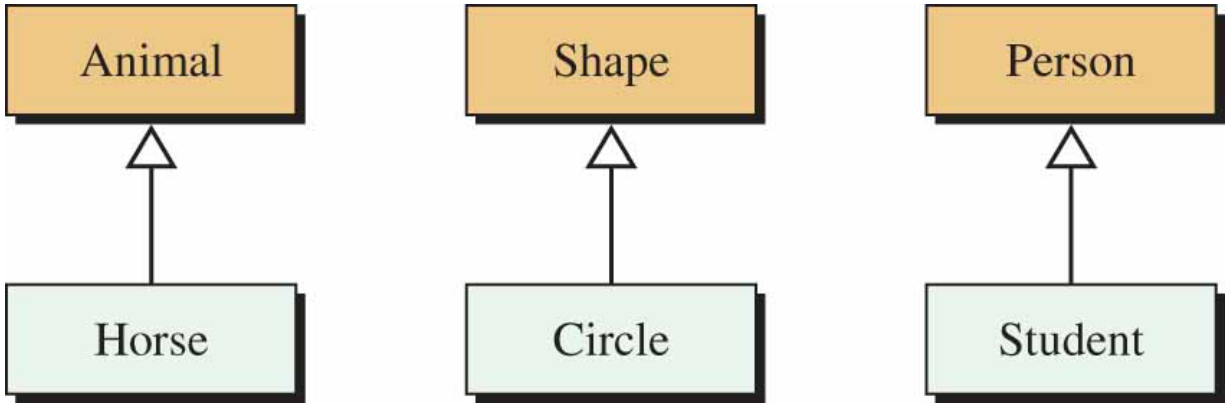
General Idea Part 1

In object-oriented programming, classes are not used in isolation.

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

To show the relationship between classes in inheritance, we use the Unified Modeling Language (UML).

UML diagram for inheritance



General Idea Part 2

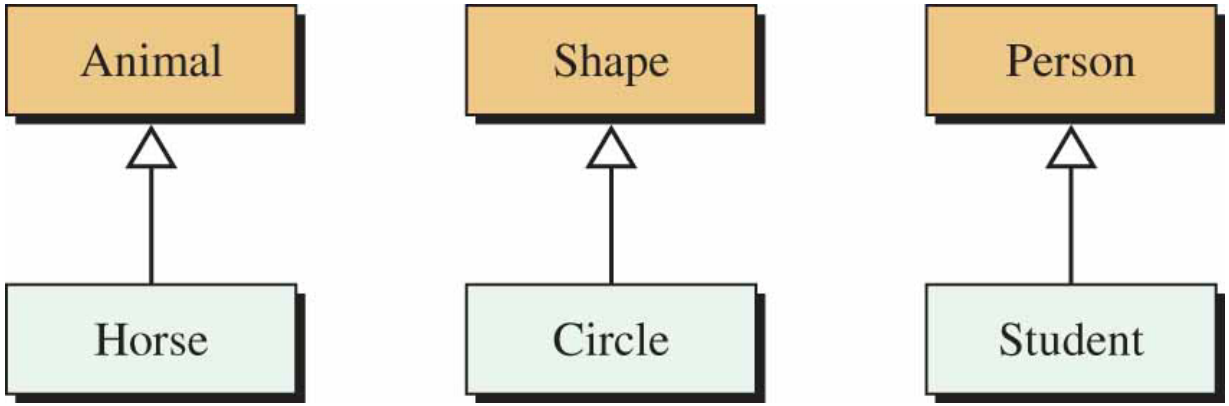
The classes are shown as rectangle boxes in UML.

The inheritance relation is shown by a line ending in a hollow triangle that goes from the more specific class to a more general class.

In C++, the most general class is called the *base class* and a more specific class is called the *derived class*.

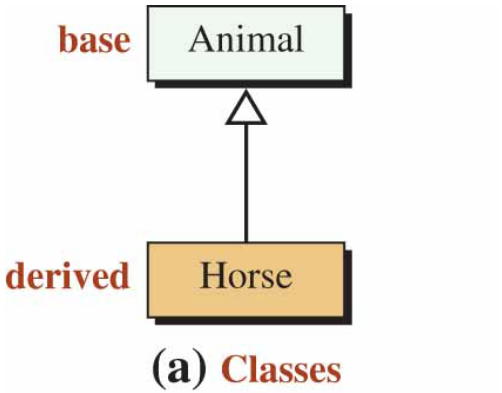
A more general class is also known as a superclass; a more specific class is also know as a subclass.

UML diagram for inheritance



Inheritance

Classes and objects in inheritance hierarchy



(a) Classes

It should be obvious that a specific concept needs to have the characteristics of the general concept, but it can have more.

That is why C++ says that a derived class extends its base class.

The term extends here means the derived class needs to have all of the data members and member functions defined in the base class (with the exception of constructors, destructor, and assignment operators that need to be redefined), and it can create new data members and member functions.

The derived class inherits all members (with some exceptions) from the base class, and it can add to them.

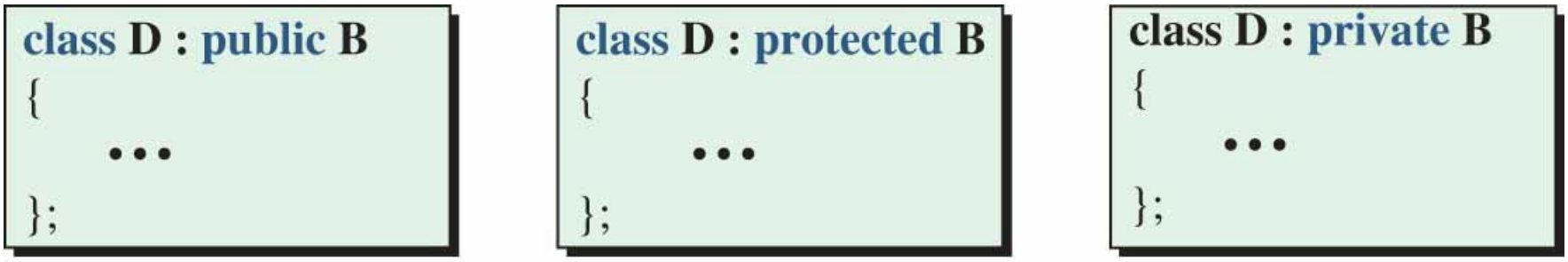
Inheritance Types

To create a derived class from a base class, we have three choices in C++: *private inheritance*, *protected inheritance*, *public inheritance*

To show the type of the inheritance we want to use, we insert a colon after the class, followed by one of the keywords (*private*, *protected*, or *public*).

Figure shows these three types of inheritance in which B is the base class and D is the derived class.

Inheritance types



Although the default inheritance type is private inheritance, the most common, by far, is public inheritance. The other two types of inheritance are rarely used.

The most common use of inheritance is public inheritance.

Public Inheritance

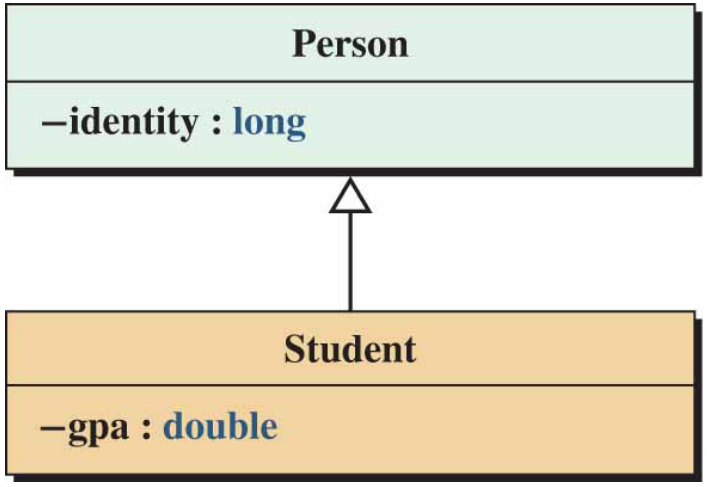
We design two classes, *Person* and *Student*, in which the class Student inherits from the class Person. We know that a student is a person.

We assume that the Person class uses only one data member: *identity*.

We also assume that the Student class needs two data members: *identity* and *gpa*.

However, since the *identity* data member is already defined in the class *Person*, it does not need to be defined in the class *Student* because of inheritance.

Two classes in inheritance relationship



Notes:

The type of data members is shown after the member name separated by a colon.

The minus signs define the visibility of data members as private.

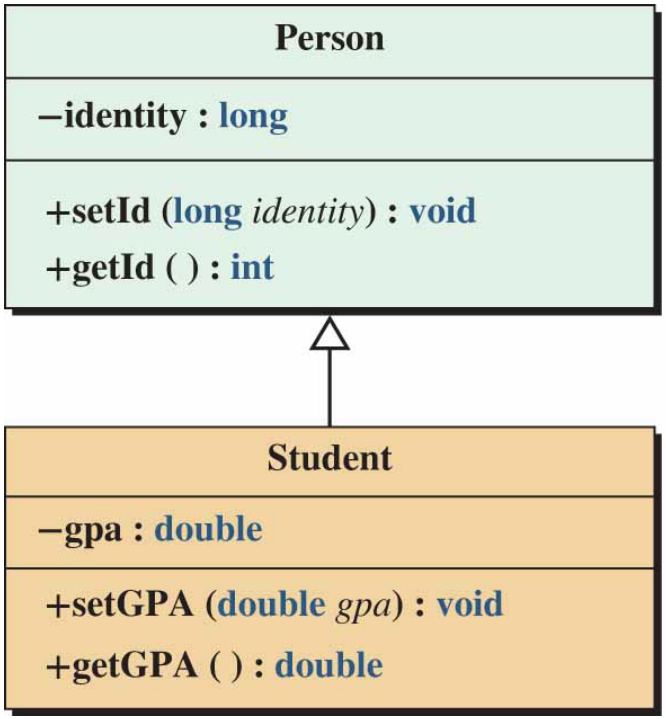
We can immediately see the advantage of inheritance.

The Student class uses the data member of the Person class and adds one data member of its own.

Public Inheritance

We add some member functions to the two classes. We ignore the constructors and destructor at this point because they are not inherited; we assume the synthetic ones are used. We add one *getter* and one *setter* function to each class.

Classes with data members and member functions



Notes:

The type of data members and member functions is shown after the member name separated by a colon.

The minus signs define the visibility of data members as private; the plus signs define the visibility of the member fuctions as public.

Exercise #1/3

Public Inheritance Part 1

```
1  /*************************************************************
2   * The program shows how we can let the class Student inherit  *
3   * from the class Person because a student is a person.        *
4   *************************************************************/
5
6  #include <iostream>
7  #include <cassert>
8  #include <string>
9  using namespace std;
10
11 /*************************************************************
12  * The class definition for the Person class                   *
13  *************************************************************/
14 class Person
15 {
16      private:
17          long identity;
18      public:
19          void setId (long identity);
20          long getId( ) const;
```

Public Inheritance Part 2

```
21  };
22  /*************************************************************
23   * The definition of setId function in the Person class      *
24   *************************************************************/
25  void Person :: setId (long id)
26  {
27       identity = id;
28       assert (identity >= 100000000 && identity <= 999999999) ;
29  }
30
31  /*************************************************************
32   * The definition of the getId function in the Person class  *
33   *************************************************************/
34  long Person :: getId () const
35  {
36       return identity;
37  }
38
39  /*************************************************************
40   * The class definition for the Student class                *
```

Public Inheritance Part 3

```
41  ************************************************************/
42
43  class Student : public Person
44  {
45       private:
46       double gpa;
47       public:
48       void setGPA (double gpa);
49       double getGPA () const;
50  };
51
52  /************************************************************
53   * The definition of setGPA function in Student class       *
54   ************************************************************/
55  void Student :: setGPA (double gp)
56  {
57       gpa = gp;
58       assert (gpa >=0 && gpa <= 4.0);
59  }
60
```

Public Inheritance Part 4

```
61  /*************************************************************
62   * The definition of getGPA function in Student class        *
63   *************************************************************/
64  double Student :: getGPA() const
65  {
66       return gpa;
67  }
68
69
70  /*************************************************************
71   * The application function (main) that uses both classes    *
72   *************************************************************/
73  int main ( )
74  {
75       // Instantiation and use of a Person object
76       Person person;
77       person.setId (111111111L);
78       cout << "Person Information: " << endl;
79       cout << "Person's identity: " << person.getId ( );
80       cout << endl << endl;
```

Public Inheritance Part 5

```
81      // Instantiation and use of a Student object
82      Student student;
83      student.setId (222222222L);
84      student.setGPA (3.9);
85      cout << "Student Information: " << endl;
86      cout << "Student's identity: " << student.getId() << endl;
87      cout << "Student's gpa: " << student.getGPA();
88      return 0;
89  }
```

```
Run:
Person Information:
Person's Identity: 111111111

Student Information:
Student's identity: 222222222
Student's gpa: 3.9
```

Private Data Members

As Figure shows, the base class object has only one data member, but the derived class object has two data members: one inherited and one created.

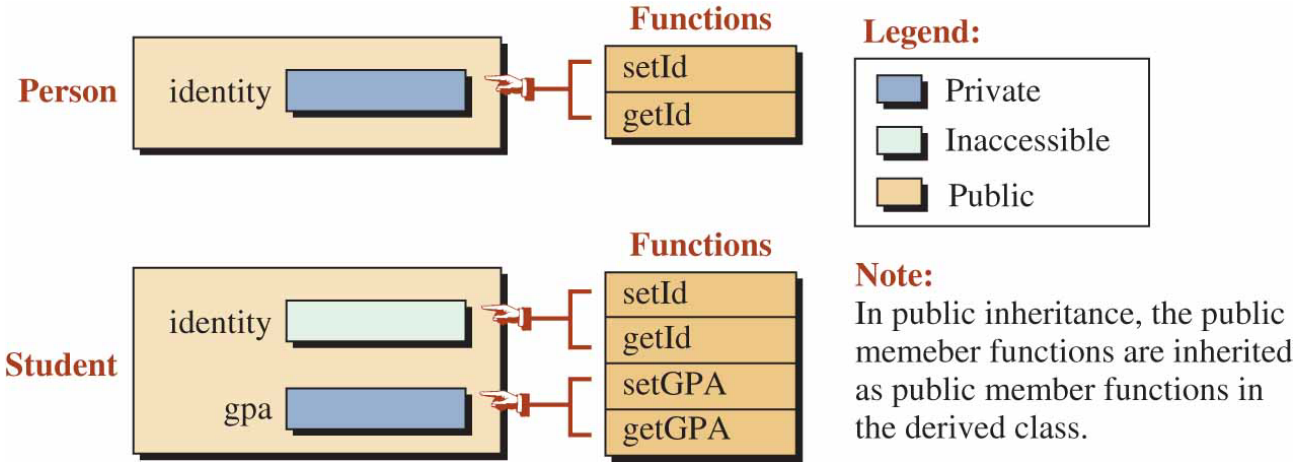
Private data members in public inheritance

A private member in the base class becomes an inaccessible (hidden) member in the derived class.

Public Member Functions

Figure adds public member functions. We can see that the base class has only two member functions while the derived class has four member functions, two inherited and two newly defined in the derived class.

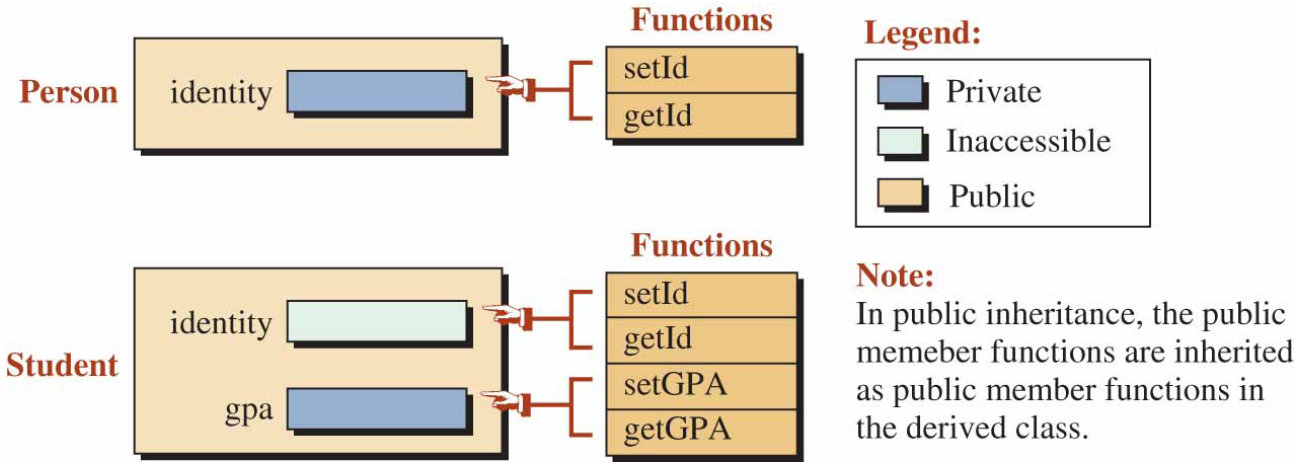
Public member functions in public inheritance



A public member in the base class becomes a public member in the derived class.

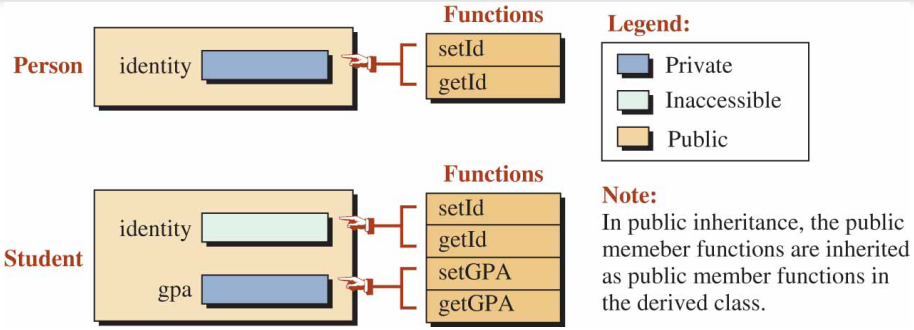
Accessing Private Data Members

Now let us show how we access private data members in each class:



- **In the base class, we can access the private data members through the public member function defined in this class (*setId* and *getId*).**
- **In the derived class, we need two groups of public member functions:**
  - **a. To access the inherited private data members, which are hidden, we use the public member functions defined in the base class (*setId* and *getId*).**
  - **b. If we need to access a data member defined in the derived class, we use the member functions defined in the derived class (*setGPA* and *getGPA*).**

Functions with Same Names in Different Classes



In the two previous classes, Person and Student, we have used two member functions for setting data members (*setId* and *setGPA*) and two member functions for getting data members (*getId* and *getGPA*).

Can we use two functions with the same name, one in the base class and the other in the derived class?

- **In other words, can we have a function named *set* in the base class and another function also named *set* in the derived class?**
- **Similarly, can we have a function named *get* in the base class and another function also named *get* in the derived class?**

The answer to both questions is positive, but we need to use two different concepts: *overloaded* and *overridden* functions.

To use the same name for a function in the base and derived classes, we need overloaded or overridden member functions.

Overloaded and Overridden Member Functions

Overloaded functions are two functions with the same name but with two different signatures.

Overloaded functions can be used in the same or different classes without being confused with each other.

We can have two functions named *set*, one in the base class and the other in the derived class with the following prototypes:

```
// In Person class
void set(long identity);
```

```
// In Student class
void set(double gpa);
```

If the signature of the two functions with the same name is the same, we have overridden member functions as shown below.

```
// In Person class
long get();
```

```
// In Student class
double get();
```

Class Scope

```
// In Person class
void set(long identity);
```

```
// In Student class
void set(double gpa);
```

```
// In Person class
long get();
```

```
// In Student class
double get();
```

How can the system distinguish which function we are calling?

We need to remember that we do not call member functions in a class just by name, we let the instance call the appropriate function.

```
// Using Person object
person.set(111111111L);
person.get();
```

```
// Using Student object
student.set(3.7);
student.get();
```

A compiler invokes a function based on the following rules:

a. **The compiler tries to find a matching function (using names and parameters) that belongs to the class of the object that has invoked the function.**

b. **If no matching function is found, the compiler looks at the functions inherited from the superclass.**

c. **If no match is found, the search continues until the base class is reached.**

Delegation Of Duty

An overloaded or overridden member function in a derived class can delegate part of its operation to a member function in a class in a higher level by calling the corresponding member function.

```
// Using Person object
void Person :: set(long id)
{
    identity = id;
}

void Person :: print()
{
    cout << name;
}
```

```
// Using Student object
void Student :: set(long id, double gp)
{
    Person :: set(id);  // Delegation
    gpa = gp;
}
void Student :: print()
{
    Person :: print();  // Delegation
    cout << gpa;
}
```

Members Not Inherited : Invocation

There are five member functions that are not inherited in the derived class:

1. **default constructor**
2. **parameter constructor**
3. **copy constructor**
4. **destructor**
5. **assignment operator.**

We postpone the discussion of the assignment operator until a future chapter, but we discuss the other four in this chapter.

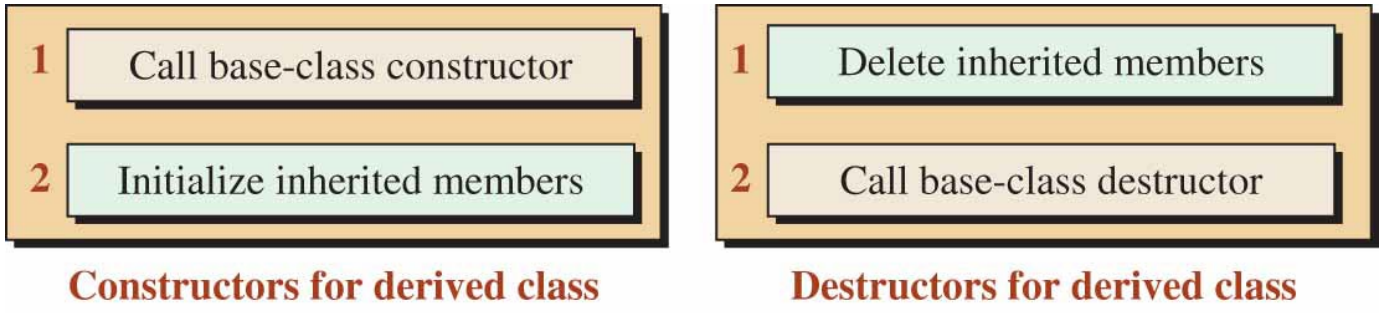
Constructors, destructor, and assignment operators are not inherited; they need to be redefined.

Constructor and Destructor in Inheritance

The constructor of the derived class cannot initialize the data members of the base class because they are hidden in the derived class.

Similarly, the destructor of a derived class cannot delete the data members of the base class because they are hidden in the derive class.

Constructor and destructor in inheritance



The constructor of the derived class invokes the constructor of the base class in its initialization and then initializes the data members of the derived class.

Similarly, the destructor of the derived class first deletes the data members of the derived class and then calls the destructor of the base class.

Note that the order of activities in a constructor and destructor are reverse of each other.

Definition of Constructors and Destructor

Base Class	Derived Class

// Default Constructor
Person :: Person()
: identity(0)
{
}
	// Default Constructor
	Student :: Student()
	: Person(0), gpa(0.0)
	{
	}

// Parameter Constructor
Person :: Person (long id)
identity(id),
{
}
	// Parameter Constructor
	Student :: Student (long id, double gp) :
	: Person(id), gpa(gp)
	{
	}

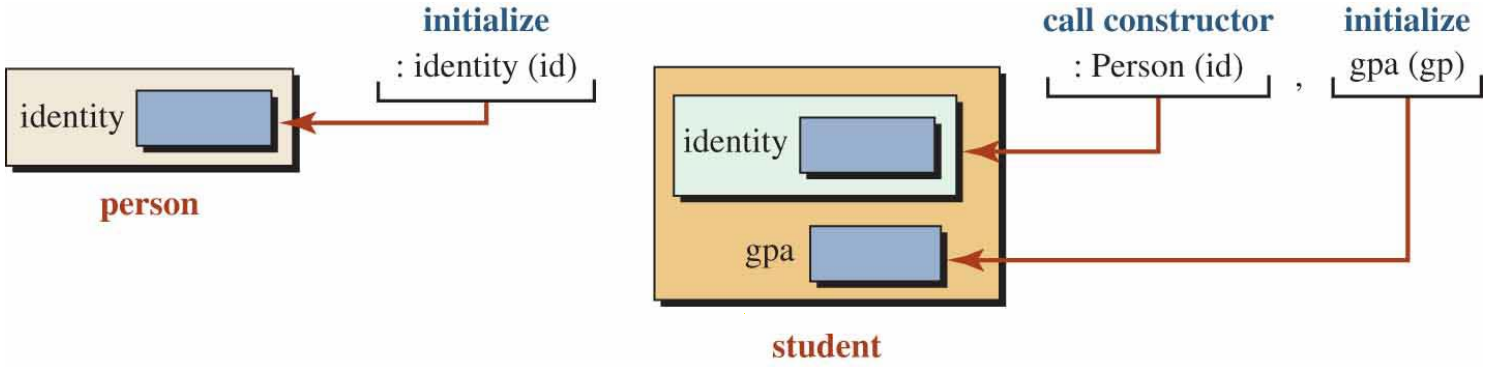
// Copy Constructor
Person :: Person(const Person& obj)
: identity(obj.identity) :
{
}
	// Copy Constructor
	Student :: Student(const Student& st)
	: Person(st) , gpa(std.gpa)
	{
	}

// Destructor
Person :: ~Person()
{
}
	// Destructor
	Student :: ~Student()
	{
	}

Constructing Objects

Figure shows how the base class and the derived class objects are constructed using the parameter constructor.

Constructing objects of type Person and Student




```
// Parameter Constructor
Person :: Person (long id)
identity(id),
{
}
```

```
// Parameter Constructor
Student :: Student (long id, double gp) :
: Person(id), gpa(gp)
{
}
```

Delegation Versus Invocation

Delegation and invocation are different concepts and are done differently.

In delegation, a derived member function delegates part of its duty to the base class using the class resolution operator (::).

```
// Using Person object
void Person :: set(long id)
{
     identity = id;
}
```

```
// Using Student object
void Student :: set(long id, double gp)
{
     Person :: set(id);  // Delegation
     gpa = gp;
}
```

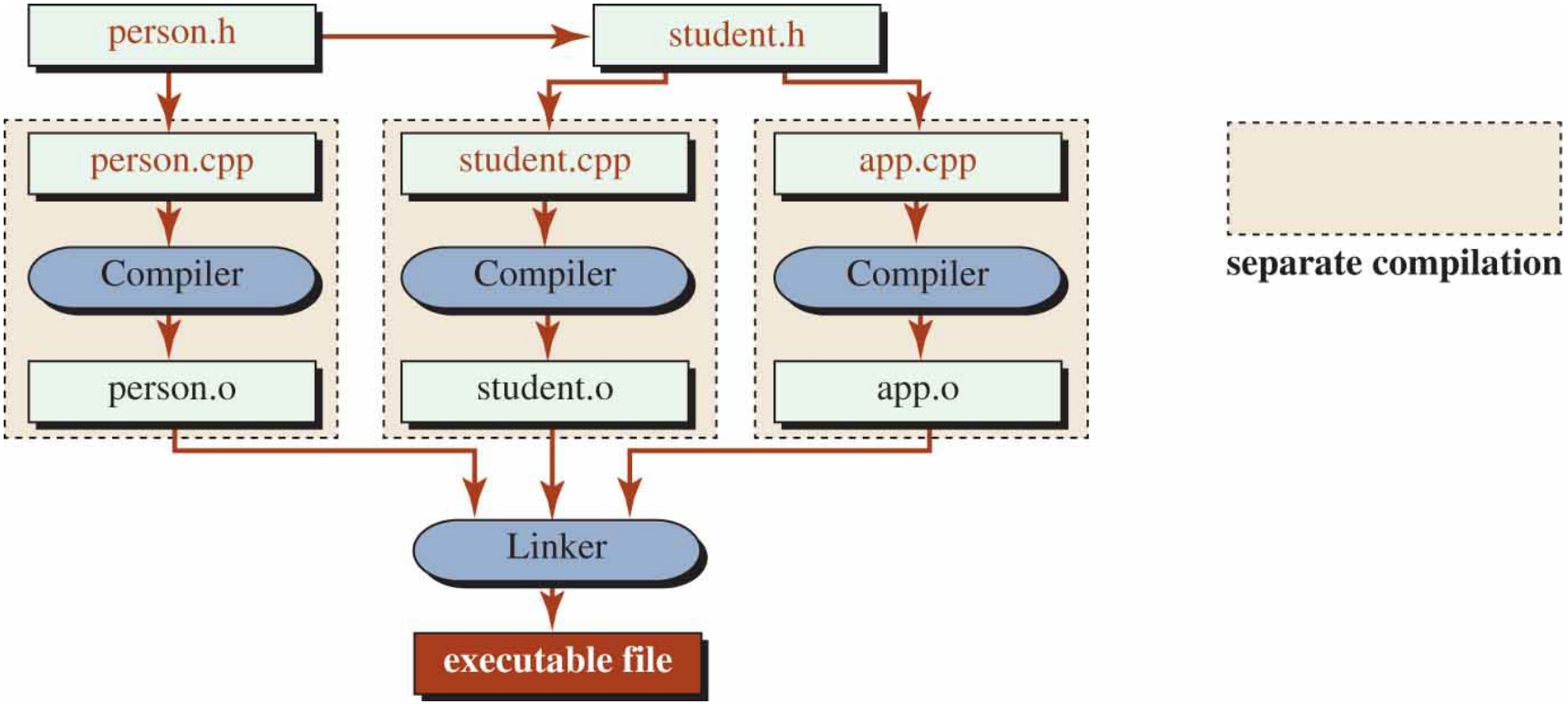
In invocation, the constructor of a derived class calls the constructor of the base class during initialization, which does not require the class resolution operator.

```
// Parameter Constructor
Person :: Person (long id)
identity(id),
{
}
```

```
// Parameter Constructor
Student :: Student (long id, double gp) :
: Person(id), gpa(gp)
{
}
```

Separate Compilation

Figure shows how separate compilation work using inheritance.



Exercise #2/3

Using Separate Compilation Part 1

```
1  /*************************************************************
2   * The interface file for the Person class                   *
3   *************************************************************/
4
5  #ifndef PERSON_H
6  #define PERSON_H
7  #include <cassert>
8  #include <iostream>
9  #include <iomanip>
10 using namespace std;
11
12 class Person
13 {
14      private:
15          long identity;
16      public:
17          Person ();
18          Person (long identity);
19          ~Person();
20          Person (const Person& person);
21          void print () const;
22 };
23 #endif
```

Using Separate Compilation Part 2

```
1  /*****************************************************************
2   * The implementation file for the Person class                  *
3   *****************************************************************/
4  #include "person.h"
5
6  // Default constructor
7  Person :: Person ()
8  : identity (0)
9  {
10 }
11 // Parameter constructor
12 Person :: Person (long id)
13 : identity (id)
14 {
15     assert (identity >= 100000000 && identity <= 999999999);
16 }
17 // Copy constructor
18 Person :: Person (const Person& person)
19 : identity (person.identity)
20 {
```

Using Separate Compilation Part 3

```
21  }
22  // Destructor
23  Person:: ~Person()
24  {
25  }
26  // Accessor member function
27  void Person :: print () const
28  {
29      cout << "Identity: " << identity << endl;
30  }
```

Interface File for Student Class

```
1  /*****************************************************************
2   * The interface file for the Student class                      *
3   *****************************************************************/
4  #ifndef STUDENT_H
5  #define STUDENT_H
6  #include "person.h"
7
8  class Student: public Person
9  {
10      private:
11          double gpa;
12      public:
13          Student ( );
14          Student (long identity, double gpa);
15          ~Student();
16          Student (const Student& student);
17          void print () const;
18 };
19 #endif
```

Implementation Files for Student Class Part 1

```
1  /*************************************************************
2   * The implementation file for the Student class             *
3   *************************************************************/
4  #include "student.h"
5
6  // Default constructor
7  Student :: Student ()
8  : Person (), gpa (0.0)
9  {
10 }
11 // Parameter constructor
12 Student :: Student (long id, double gp)
13 : Person (id), gpa (gp)
14 {
15     assert (gpa >= 0.0 && gpa <= 4.0);
16 }
17 // Copy constructor
18 Student :: Student (const Student& student)
19 : Person (student), gpa (student.gpa)
20 {
```

Implementation Files for Student Class Part 2

```
21  }
22  // Destructor
23  Student :: ~Student()
24  {
25  }
26  // Accessor member function
27  void Student ::print () const
28  {
29      Person :: print ();
30      cout << "GPA: " << fixed << setprecision (2) << gpa << endl;
31  }
```

Application File to Use the Classes

```
1  /*************************************************************
2   * The application to test the Person and Student classes     *
3   *************************************************************/
4  #include "student.h"
5
6  int main ( )
7  {
8       // Instantiation and using a Person object
9       Person person (111111111);
10      cout << "Information about person: " << endl;
11      person.print ();
12      cout << endl;
13      // Instantiation and using a Student object
14      Student student (222222222, 3.9);
15      cout << "Information about student: " << endl;
16      student.print ();
17      cout << endl;
18      return 0;
19 }
```

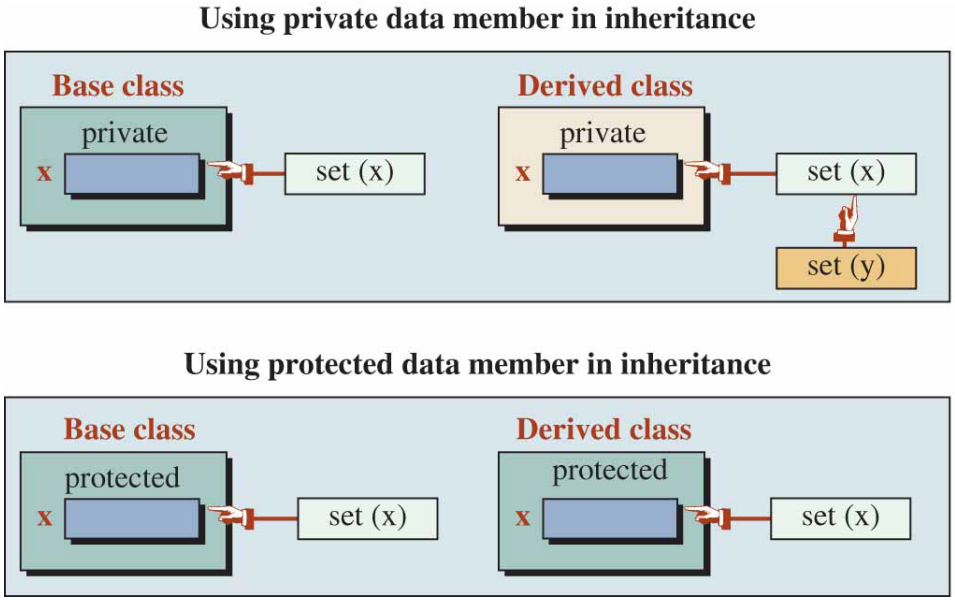
```
Run:
Information about person:
Identity: 111111111

Information about student:
Identity: 222222222
GPA: 3.90
```

More About Public Inheritance Part 1

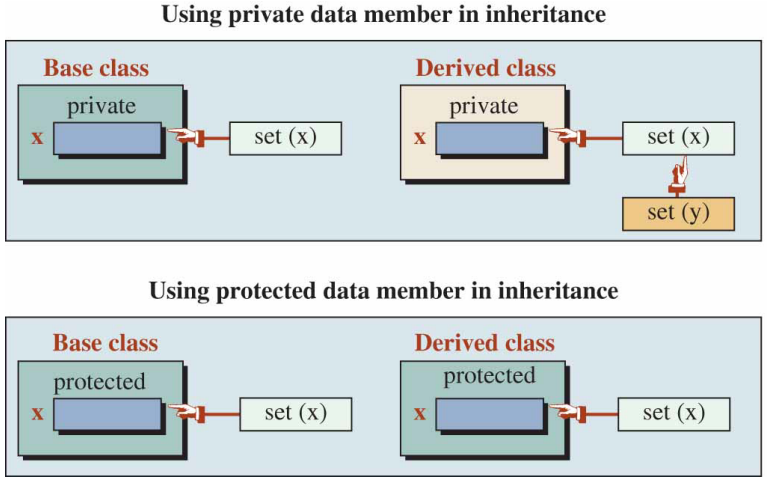
Protected Members

C++ also defines another specifier for a member, *protected*. A protected member is accessible in the derived class and all classes derived from the derived class.

The functions defined in the derived class can directly access a protected member in base class.



More About Public Inheritance Part 2



Note:
Parameter y is a dummy one to be used to set x.

We have only one private data member and one public member function.

When we use derivation, the private member *x* is hidden.

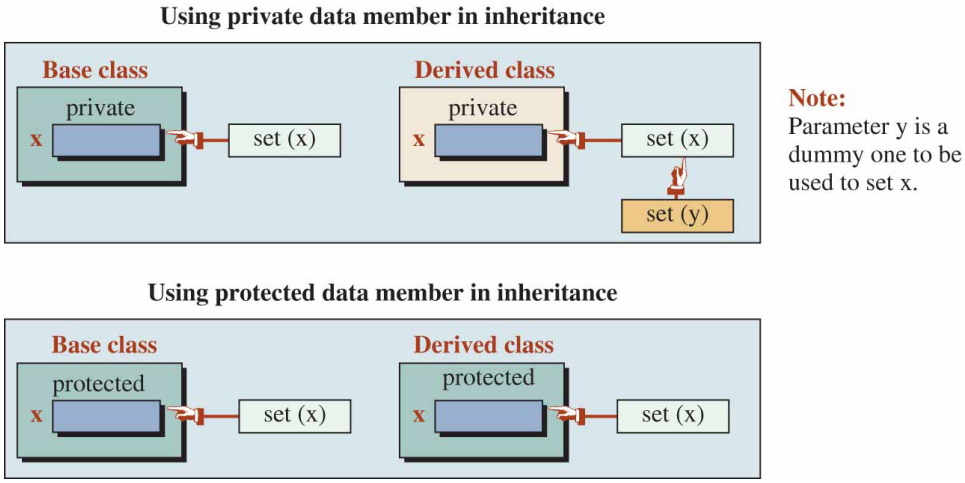
We need to create another function, with another dummy variable *y* to access *x* as shown below:

```
void Derived :: set(int y)
{
     Base :: set(y);    // Calling the inherited function
}
```

If we have defined the data member as protected, it is seen in the derived class and we can use set(*x*) directly without having to create another function with a dummy variable.

```
void Derived :: set(int y)
{
     this->x = y;    // Accessing directly a protected member
}
```

Using Private Data Members



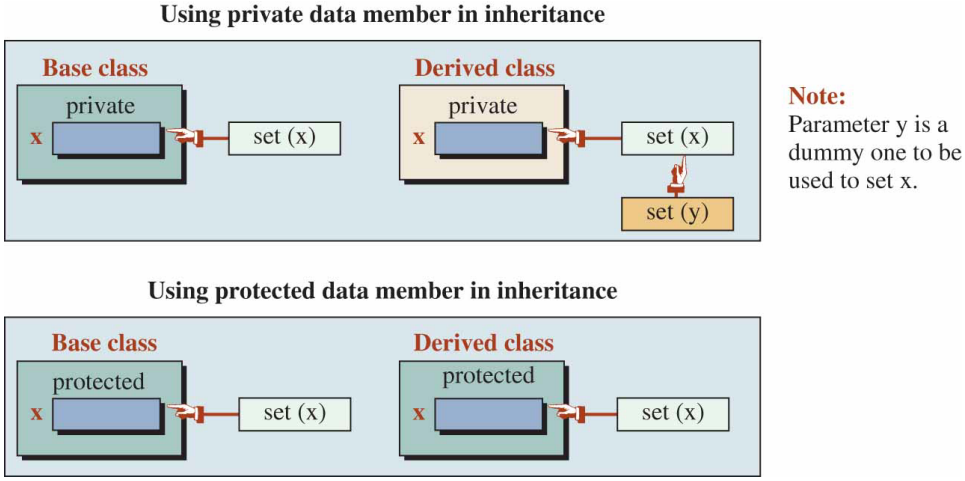
Using private data members enforces the concept of encapsulation.

When we use private data members, data are hidden to entities outside of the class and also to derived classes.

On the other hand, using private data members means creating more code in the derived class.

Therefore, the advantage of private data members is stronger encapsulation; its disadvantage is creating more code in the derived class.

Using Protected Data Members



Using protected data members makes the coding of the derived classes simpler.

However, it breaks the idea of encapsulation.

Sometimes we need to use protected data members because the coding becomes very complicated if we use private data members.

Blocking Inheritance

Sometimes we may want to block inheritance. The C++ standard allows us to do so using the final modifier as shown below:

```
class First final
{
    …
}
```

We can also use the final modifier to stop the inheritance anywhere in the hierarchy.

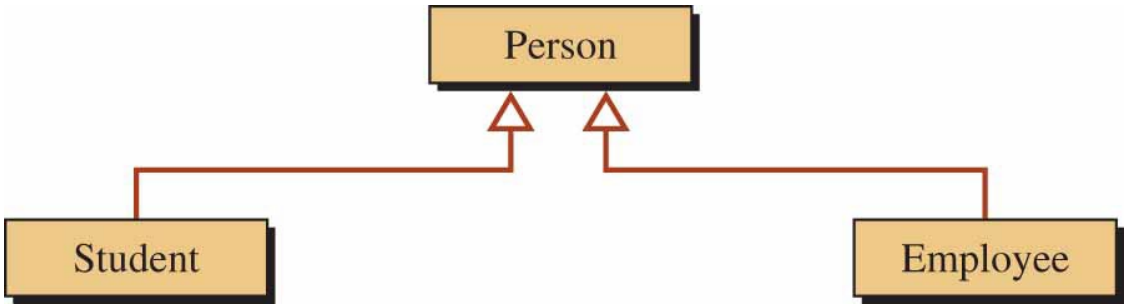
```
class First
{
    …
}
class Second final : public First
{
    …
}
```

Inheritance Tree

In C++, we can have an inheritance tree. For example, we can have two classes that inherit from the Person class: Student and Employee.

It is clear that a student is a person and an employee is also a person.

Inheritance Tree



Exercise #3/3

Interface File to Define the Person Class Part 1

```
1  /*************************************************************
2   * The interface file to define the Person class             *
3   *************************************************************/
4  #ifndef PERSON_H
5  #define PERSON_H
6  #include <iostream>
7  #include <string>
8  using namespace std;
9
10 // Definition of Person class
11 class Person
12 {
13      private:
14          string name;
15      public:
16          Person (string nme);
17          ~Person();
18          void print () const;
19 };
20 #endif
```

Implementation File for the Person Class

```
1  /*************************************************************
2   * The implementation file for the Person class              *
3   *************************************************************/
4  #include "person.h"
5
6  // Constructor for Person class
7  Person :: Person (string nm)
8  :name (nm)
9  {
10 }
11 // Destructor for Person class
12 Person :: ~Person()
13 {
14 }
15 // Definition of print member function
16 void Person :: print () const
17 {
18      cout << "Name: " << name << endl;
19 }
```

Interface File to Define the Student Class

```
1  /*****************************************************************
2   * The interface file to define the Student class                *
3   *****************************************************************/
4  #ifndef STUDENT_H
5  #define STUDENT_H
6  #include "person.h"
7
8  // Definition of the Student class
9  class Student : public Person
10 {
11      private:
12          string name;
13          double gpa;
14      public:
15          Student (string name, double gpa);
16          ~Student ( );
17          void print () const;
18 };
19 #endif
```

Implementation File for the Student Class

```
1  /*************************************************************
2   * The interface file to define the Employee class           *
3   *************************************************************/
4  #include "student.h"
5
6  // Constructor for Student class
7  Student :: Student (string nm, double gp)
8  :Person (nm), gpa (gp)
9  {
10 }
11 // Destructor for Student class
12 Student :: ~Student ()
13 {
14 }
15 // Definition of the print member function
16 void Student :: print () const
17 {
18      Person :: print();
19      cout << "GPA: " << gpa << endl;
20 }
```

Interface File to Define the Employee Class

```
1  /*****************************************************************
2   * The interface file to define the Student class                *
3   *****************************************************************/
4  #ifndef EMPLOYEE_H
5  #define EMPLOYEE_H
6  #include "person.h"
7
8  // Definition of the Employee class
9  class Employee : public Person
10 {
11      private:
12          string name;
13          double salary;
14      public:
15          Employee (string name, double salary);
16          ~Employee ( );
17          void print () const;
18 };
19 #endif
```

Implementation File for the Employee Class

```
1  /*************************************************************
2   * The implementation file for the Employee class            *
3   *************************************************************/
4  #include "employee.h"
5
6  // Constructor for Employee class
7  Employee :: Employee (string nm, double sa)
8  :Person (nm), salary (sa)
9  {
10 }
11 // Destructor for Employee class
12 Employee :: ~Employee()
13 {
14 }
15 // Definition of print member function
16 void Employee :: print () const
17 {
18      Person :: print();
19      cout << "Salary: " << salary << endl;
20 }
```

Application File to Use Classes Part 1

```
1  /*************************************************************
2   * The application file to use classes                       *
3   *************************************************************/
4  #include "student.h"
5  #include "employee.h"
6
7  int main ()
8  {
9       // Instantiation and using an object of the Person class
10      cout << "Person: " << endl;
11      Person person ("John");
12      person.print ();
13      cout << endl << endl;
14      // Instantiation and using an object of the Student class
15      cout << "Student: " << endl;
16      Student student ("Mary", 3.9);
17      student.print ();
18      cout << endl << endl;
19      // Instantiation and using an object of the Employee class
20      cout << "Employee: " << endl;
```

Application File to Use Classes Part 2

```
21      Employee employee ("Juan", 78000.00);
22      employee.print ();
23      cout << endl << endl;
24      return 0;
25  }
```

```
Run:
Person:
Name: John

Student:
Name: Mary
GPA: 3.9

Employee:
Name: Juan
Salary: 78000
```

KU KONKUK UNIVERSITY

What's Next?

Reading Assignment

- **Read Chap. 11. Relationships among Classes**

End of Class

Thank you

E-mail: youngcha@konkuk.ac.kr